

Java™ magazine

By and for the Java community



//JULY/AUGUST 2014 /

THE JAVA VIRTUAL MACHINE ALTERNATIVE LANGUAGES

Explore the flexibility of the JVM
with the language of your choice

05

ALTERNATIVE LANGUAGES FOR THE JAVA VIRTUAL MACHINE

A look at eight features from eight JVM languages



New theme icon. [See how it works.](#)

COMMUNITY

03

From the Editor

12

Java Nation

News, people, books, and events

JAVA IN ACTION

24

Java Gets Its Wings

Russian research institute creates custom air traffic control system with NetBeans and Java.

JAVA TECH

29

New to Java
Interactive Objects with BlueJ

Visualization and interaction tools illustrate aspects of object-oriented programming.

33

Java Architect
Performing 10 Routine Operations Using Different JVM Languages

Venkat Subramaniam explores the unique capabilities of Groovy, Java, and Scala.

41

Java Architect
Avoiding Benchmarking Pitfalls on the JVM

Use JMH to write useful benchmarks that produce accurate results.

50

Java Architect
Understanding Java JIT Compilation with JITWatch

Ben Evans provides a primer on JIT compilation in Java HotSpot VM.

18

CREATE THE FUTURE AT JAVAONE

See what's on the lineup at this year's JavaOne.

53

Enterprise Java

PRODUCTION-TIME PROFILING WITH ORACLE JAVA MISSION CONTROL

Low-overhead profiling and diagnostics for Java applications running on the JVM

57

Enterprise Java
Server-Side Deployment on the JVM

JVM deployments with the server outside, inside, and alongside a web app

61

Enterprise Java
Introduction to the Java Temporary Caching API

Use a caching strategy without worrying about implementation details.

67

Rich Client
JavaFX with Alternative Languages

Alternative languages and custom APIs enable rapid JavaFX application development.

76

Fix This

Take our generics code challenge!

EDITORIAL

Editor in Chief

[Caroline Kvitka](#)

Community Editor

[Yolande Poirier](#)

Java in Action Editor

[Michelle Kovac](#)

Technology Editor

[Tori Wieldt](#)

Contributing Writer

[Kevin Farnham](#)

Contributing Editors

[Claire Breen](#), [Blair Campbell](#),
[Kay Keppler](#), [Karen Perkins](#)

DESIGN

Senior Creative Director

[Francisco G Delgadillo](#)

Senior Design Director

[Suemi Lam](#)

Design Director

[Richard Merchán](#)

Contributing Designers

[Jaime Ferrand](#), [Arianna Pucherelli](#)

Production Designers

[Sheila Brennan](#), [Kathy Cygnarowicz](#)

PUBLISHING

Publisher

[Jennifer Hamilton](#) +1.650.506.3794

Associate Publisher and Audience Development Director

[Karin Kinnear](#) +1.650.506.1985

ADVERTISING SALES

President, Sprocket Media

[Kyle Walkenhorst](#) +1.323.340.8585

Western and Central US, LAD, and Canada, Sprocket Media

[Tom Cometa](#) +1.510.339.2403

Eastern US and EMEA/APAC, Sprocket Media

[Mark Makinney](#) +1.805.709.4745

Advertising Sales Assistant

[Cindy Elhaj](#) +1.626.396.9400 x 201

Mailing-List Rentals

Contact your sales representative.

RESOURCES

Oracle Products

+1.800.367.8674 (US/Canada)

Oracle Services

+1.888.283.0591 (US)

Oracle Press Books

[oraclepressbooks.com](#)

COMMUNITY

JAVA IN ACTION

JAVA TECH

ABOUT US



ARTICLE SUBMISSION

If you are interested in submitting an article, please [e-mail the editors](#).

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the [subscription form](#).

MAGAZINE CUSTOMER SERVICE

[java@halldata.com](#) **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or e-mail address not be included in this program, contact [Customer Service](#).

Copyright © 2014, Oracle and/or its affiliates. All Rights Reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the editors. JAVA MAGAZINE IS PROVIDED ON AN "AS IS" BASIS. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DAMAGES OF ANY KIND ARISING FROM YOUR USE OF OR RELIANCE ON ANY INFORMATION PROVIDED HEREIN. The information is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle. Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Java Magazine is published bimonthly with a free subscription price by Oracle, 500 Oracle Parkway, MS OPL-3C, Redwood City, CA 94065-1600.

Digital Publishing by [GTxcel](#)



SEPTEMBER 28 – OCTOBER 2, 2014
SAN FRANCISCO

[oracle.com/javaone](#)

REGISTER NOW

Respond Early, Save \$400*

Discover All Things Java

- Learn all about Java 8
- 500+ educational sessions
- 100+ product demos
- Meet with 500+ Java experts



Diamond Sponsor



Silver Sponsor



Bronze Sponsors



*Save \$400 over onsite registration. Visit [oracle.com/javaone/register](#) for early bird registration dates and details.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

ORACLE®

//from the editor /



W

hile often overshadowed by the **Java language**, the Java Virtual Machine (JVM) is the cornerstone of the Java platform. It gives Java its hardware and operating system independence, small compiled code size, and protection from malicious programs. We've dedicated almost an entire issue to the JVM, so if you haven't paid much attention to it lately, buckle up and get ready for the ride.

One of the most notable things about the JVM is that it can run hundreds of programming languages besides the Java language—providing developers with flexibility and freedom of choice. In our cover story, "[Alternative Languages for the JVM](#)," Raoul-Gabriel Urma takes a look at eight JVM languages and some of the benefits of using them. In "[Performing IO Routine Operations Using Different JVM Languages](#)," Venkat Subramaniam further explores the benefits of alternative languages—while also showing how Java SE 8 handles certain tasks. Developing JavaFX applications? Check out Josh Juneau's article, "[JavaFX with Alternative Languages](#)," to discover how alternative languages and custom APIs can speed development.

We also focus on things you can do to make the JVM work better for you. [Julien Ponge](#) shares advice on avoiding benchmarking pitfalls on the JVM, and [Cas Saternos](#) gets us up to speed on server-side deployment on the JVM. In addition, [Marcus Hirt](#) introduces us to a profiling and diagnostics tool for Java applications running on the JVM and [Ben Evans](#) gives us a primer on just-in-time compilation.

Want more? Make sure to register for [JavaOne](#), September 28–October 2 in San Francisco, California, which features a dedicated JVM Languages track (and check out our [conference preview](#)). Hope to see you there!

Caroline Kvitka, Editor in Chief 



//send us your feedback /

We'll review all suggestions for future improvements. Depending on volume, some messages might not get a direct reply.



Find the Most Qualified Java Professionals for your Company's Future

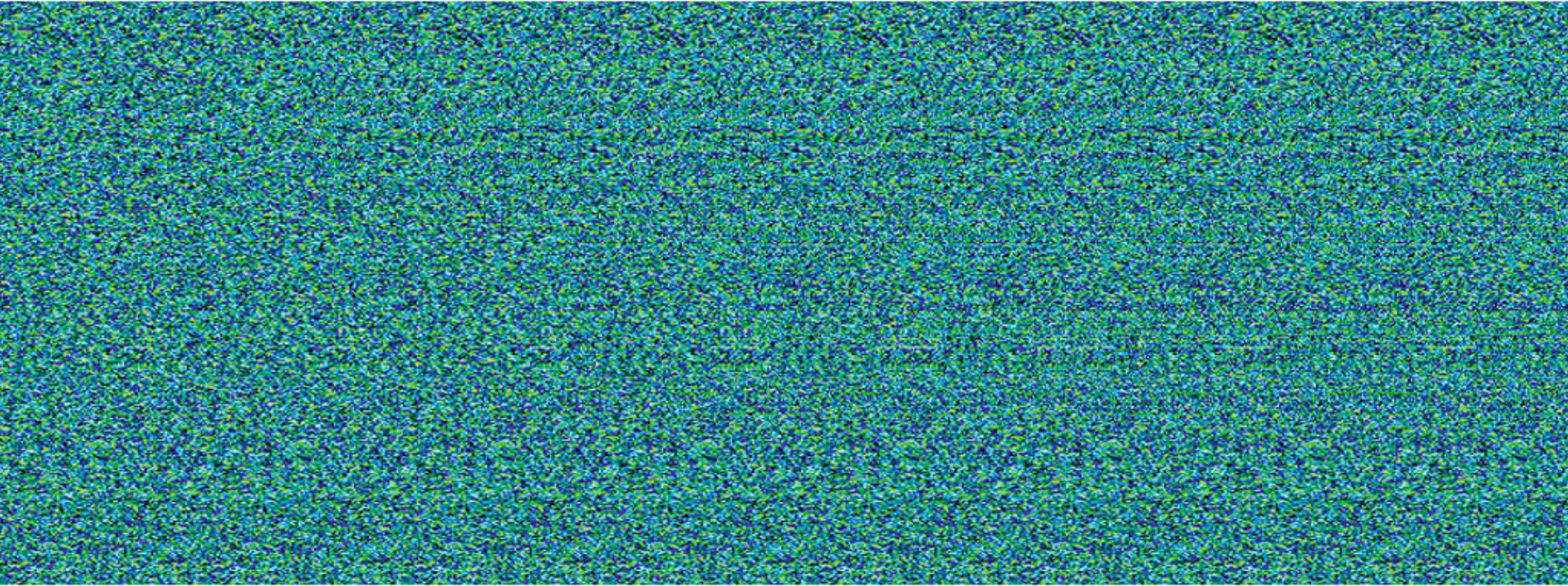
Introducing the *Java Magazine* Career Opportunities section – the ultimate technology recruitment resource.

Place your advertisement and gain immediate access to our audience of top IT professionals worldwide including: corporate and independent developers, IT managers, architects and product managers.

For more information or to place your recruitment ad or listing contact:
tom.cometa@oracle.com

PHOTOGRAPH BY BOB ADLER

The answer is right in front of you



Java Image Enabling SDKs that Help You See the Big Picture

At first glance it may seem difficult, but it's really quite simple. Atalasoft's JoltImage product is a proven SDK for image enabling your Java-based web applications, easily. Image enabling helps to add dimension to your data, so you can uncover insights such as correlations and causations hidden inside your 2-dimensional documents. Our SDK does the heavy lifting for you, saving time, money, and the headaches of figuring it out yourself. Backed by our highly knowledgeable & caffeinated support engineers, JoltImage will enable your success and make the big picture so much easier to see.



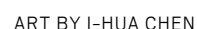
Click for tips on viewing the stereogram

Image enabling experts & bacon connoisseurs. Visit us online to see our full line of SDK products for .NET and Java



BY RAOUL-GABRIEL URMA

The JVM languages fall into three categories: They have features that Java doesn't have, they are ports of existing languages to the JVM, or they are research languages.



An abstract geometric composition featuring a dark green triangle on the left, a light green circle at the top, a medium green circle in the center, a large light green pentagon at the bottom, and a small light green square at the very bottom center.

In this macro definition, the symbol `branches` consists of a list that contains the two expressions representing

LISTING 9

Red Hat developed Ceylon, a statically typed object-oriented language, to give Java programmers a language that's easy to learn and understand (because of syntax that's similar to Java) but less verbose. Ceylon includes more type system features than Java. For example, Ceylon supports a construct for defining type aliases (similar to C's `typedef`; for example, you could define `Strings` to be an alias for `List<String>`), flow typing (for example, no need to cast the type of an expression in a block if you've already

done an `instanceof` check on it), union of types, and local type inference. In addition, in Ceylon you can ask certain variables or blocks of code to use dynamic typing—type checking is performed at runtime instead of compile time.

Feature focus: for comprehensions. `for` comprehensions can be seen as syntactic sugar for a chain of `map`, `flatMap`, and `filter` operations using Java SE 8 streams. For example, in Java, by combining a range and a `map` operation, you can generate all the numbers from 2 to 20 with a step value of `2`, as shown in **Listing 10**.

In Ceylon, it can be written as follows using a **for** comprehension:

```
List<Integer> numbers =  
[for (x in 1...10) x * 2];
```

Here's a more-complex example. In Java, you can generate a list of points in which the sum of the **x** and **y** coordinates is equal to **10**. See **Listing 11**.

Thinking in terms of `flatMap` and `map` operations using the Stream API might be overwhelming. Instead, in Ceylon, you can write more simply, as done in the code shown in **Listing 12**, which produces `[(1, 9), (2, 8), (3, 7), (4, 6), (5, 5), (6, 4), (7, 3), (8, 2), (9, 1)]`.

The result: Ceylon can make your code more concise.

6 | XTEND

Xtend is a statically typed object-oriented language. One way it differs from other languages is that it compiles to pretty-printed Java code rather than bytecode. As a result, you can also work with the generated code.

Xtend supports two forms of method invocation: default Java dispatching and multiple dispatching. With multiple dispatching, an overloaded method is selected based on the runtime type of its arguments (instead of the traditional static types of the arguments, as in Java). Xtend provides many other popular features available in other languages such as operator overloading and type inference.

One unique feature is template expressions, which are a convenient way to generate string concatenation (similar to what template engines provide). For example, template expressions support control-flow constructs such as **IF** and **FOR**. In addition, special processing of white space allows templates to be readable and their output to be nicely formatted.

Feature focus: active annotations.

Xtend provides a feature called active annotations, which is a way to do

compile-time metaprogramming. In its simplest form, this feature allows you to generate code transparently, such as adding methods or fields to classes with seamless integration in the Eclipse IDE for example. New fields or methods will show up as members of the modified classes within the Eclipse environment. More-advanced use of this feature can generate a skeleton of design patterns such as the visitor or observer pattern. You can provide your own way to generate code using template expressions.

Here's an example to illustrate this feature in action. Given sample JSON data, you can automatically generate a domain class in your Xtend program that maps JSON properties into members. The Eclipse IDE will recognize these members, so you can use features such as type checking and auto-completion. All you have to do is wrap the JSON sample within an `@Jsonized` annotation. **Figure 2** shows an example within the Eclipse IDE using a JSON sample representing a tweet.

LISTING 10

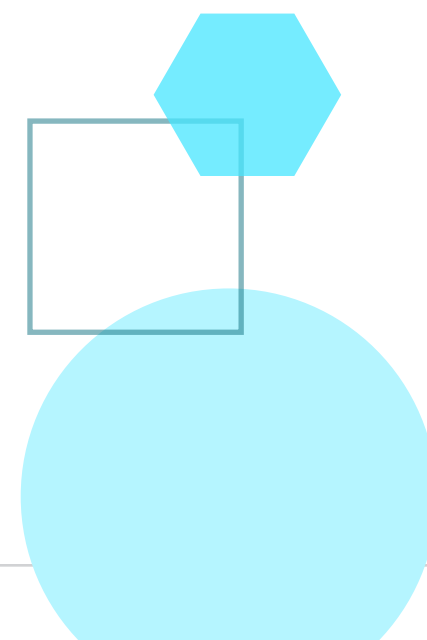
[]ava]

```
List<Integer> numbers = IntStream.rangeClosed(1, 10).mapToObj(
    x -> x * 2).collect(toList());
```

LISTING 11 / LISTING 12



[Download all listings in this issue as text](#)



Compared to popular object-oriented languages, a novel feature in its type system is support for *constraint types*. You can think of constraint types as a form of contracts attached to types. What makes this useful is that errors are checked statically, eliminating the need for more-expensive runtime checks. For example, one possible application of constraint types is to report out-of-bound array accesses at compile time.

Consider a simple **Pair** class, with a generated constructor:

You can create **Pair** objects as follows:

However, you can also define explicit constraints (similar to contracts) on the properties of a `Pair` at use-site. Here, you want to ensure that `p2` holds only symmetric pairs (that is, the values of `x` and `y` must be equal):

Because `x` and `y` are different in this code example, the assignment will be

```
const class Transaction {
  const Int value
  const Location location := Location("Cambridge")
}
class Location{
  Str city
  new make(Str city) { this.city = city }
}
```

reported as a compile error. However, the following code compiles without an error:

CONCLUSION

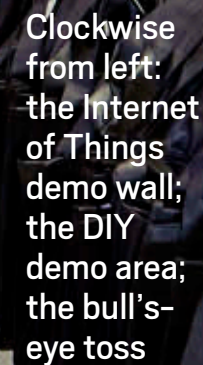
In this article, we examined eight features from eight popular JVM languages. These languages provide many benefits, such as enabling you to write code in a more concise way, use dynamic typing, or access popular functional programming features.

Acknowledgements. I'd like to thank Alex Buckley, Richard Warburton, Andy Frank, and Sven Efftinge for their feedback.



Raoul-Gabriel Urma started his PhD in computer science at the University of Cambridge at the age of 20. He is a coauthor of *Java 8 in Action: Lambdas, Streams, and Functional-Style Programming* (Manning Publications, 2014). In addition, he has given more than 20 technical talks at international conferences. He holds a MEng degree in computer science from Imperial College London and graduated with first-class honors, having won several prizes for technical innovation.

- "Java Generics Adoption: How New Features Are Introduced, Championed, or Ignored"



Java was in full swing May 17–18, 2014, in San Mateo, California, at Maker Faire, a festival of invention, creativity, and resourcefulness—and a celebration of the Maker Movement. At the Java booth, faire attendees learned how they could create the future with Java. The booth included a do-it-yourself (DIY) demo area, a bull's-eye toss, and an Internet of Things (IoT) demo wall. In addition, ORACLE TEAM USA crew member **Brad Webb** was on hand to talk about the America's Cup and sign autographs next to a replica of the winning sailboat.

12



lection of development boards running Oracle Java SE Embedded and Oracle Java ME Embedded to control lights using a guitar or a flute, ask about sensor values using Twitter and Google Voice, and call a phone number to turn on a lamp or a popcorn machine. The 4-D cube was integrated with Minecraft to make TNT explode in the game and to change the background and brightness of the Minecraft world.

MakerCon, a two-day conference and workshop for those at the forefront of the Maker Movement, was held at Oracle headquarters in Redwood Shores, California, May 13–14, 2014. The conference connected experts in digital manufacturing, technology and tools providers, accelerators that facilitate taking a prototype to market, and a broad swath of makers. Conference themes included tools of innovation, the IoT, the business side of making, and community building. An innovation showcase was also held the first evening of the event, with more than two dozen makers displaying their products and devices in a casual atmosphere. Keynote speakers included Oracle's **Jeremy Ashley** on designing new enterprise experiences for the IoT; Arduino's **Massimo Banzi**, who updated the audience on new Arduino boards and cloud services; and Autodesk's **Carl Bass**, who announced an Autodesk 3-D printer.



In the DIY area, faire attendees got hands-on with Java on workstations connected to a Raspberry Pi with a dual-sensor board that could detect proximity and luminosity. Using a simple icon-based application, users created “if-this-then-that” actions to turn on a light, move a robotic arm, turn on a table fan, send a tweet or text, take a picture, and more.

In the bull's-eye area, attendees threw foam stress balls at a smart target powered by a Raspberry Pi. If they hit the target, a sound played and a camera connected to the Raspberry Pi took a picture and uploaded it to a photo wall.

The demo wall included an IoT Java panel and a Minecraft Java IoT 4-D cube. The IoT Java panel used a col-

JAVA IN ACTION



ABOUT US



14

14

14



14

14

The Java Tour 2014 is a series of events (77 and counting) for Java developers held around the world. The Oracle Events team is producing the tour in coordination with Oracle Technology Network and Java user groups (JUGs). Oracle Java evangelists and Java Champions provide top-notch content and a chance to network with other developers. The Java Tour 2014 allows both Oracle and the Java community to address local, regional-specific Java topics. You'll hear the latest on Java 8, Java EE, and embedded Java. Check the tour page for dates and cities.



JAVA CHAMPION PROFILE

CHRISTIAN ULLENBOOM



Christian Ullenboom is a Java trainer, a best-selling author, and a blogger from Germany. He was selected to be a Java Champion in September 2005.

Java Magazine:
Where did you
grow up?

Ullenboom: I grew up in a small town in Germany—about 8,000 people. But can you really call a guy who is still fascinated about a jumping deform-ing sphere on a C64 grown up?

Java Magazine:
When and how did
you first become

interested in computers and programming?

Ullenboom: I was born in the 1970s, the beginning of the home computer era. My first computer was a C64 that had no external storage. Without the ability to save any programs, I had to type them over and over again, and so I learned programming.

Java Magazine: What was your first professional programming job?

Ullenboom: In my youth, I did a lot of performance-related Assembler demo programming (for fun, not for cash), mainly on Amiga, some on a SPARCstation at the university. After I moved from Assembler to C++, I

got a part-time job while studying, and was programming sewing machines.

Java Magazine: What do you like about the Java ecosystem?

Ulllenboom: The enormous range of open source libraries and frameworks makes Java a great choice for programming. Although I did some Python development—and I love this language, too—I never saw a compelling reason to move from Java to another language or runtime.

Java Magazine: What do you enjoy for fun and relaxation?

Ullenboom: I like to combine my work as an author with traveling around the globe. I'm writing this after a bus trip from north Luzon [Philippines]

to Manila, during which I updated my book on Java 8 features while my darling was sleeping.

Java Magazine:
Has being a Java
Champion changed
anything for you
with respect to your
daily life?

Ullенboom: Not in terms of stalkers or groupies. I became a Java Champion in 2005 because of my books on Java, which are known to almost every German-speaking Java developer. In 2013 “Java Is an Island Too” [translated title] was the most sold computer book of all IT books in Germany. However, because my books and also my blog are written in my mother language only, I am almost unknown internationally.

Java Magazine: What are you looking forward to in the coming years?

Ullenboom: Over the last years, my collection of home computers and game consoles has become very comprehensive, and I am in the process of setting up a 2,000-square-meter building for a museum where people can play with the good old stuff. Donations are welcome.

You can find
Christian Ullenboom
on Twitter ([@javabuch](#)).



EVENTS

JavaOne 2014 SEPTEMBER 28–OCTOBER 2
SAN FRANCISCO, CALIFORNIA

Top Java experts from around the world gather for this weeklong conference that includes practical hands-on content. Track topics this year range from the stronger-than-ever core Java platform to in-depth and timely explorations of Java and security, Java and the cloud, and other mission-critical Java topics. Tracks will also focus on the tools and techniques that help create outstanding user experiences that can be delivered through a variety of channels, including personal devices, smartcards, embedded environments, and intelligent equipment.

PHOTOGRAPH BY JOE PARKS AT FLICKR

OSCON

JULY 20–24
PORTLAND, OREGON

The Open Source Convention (OSCON) is an immersive five days of all things open source: new and innovative projects, major enterprise-wide deployments, and—from icons of the open source movement—deep perspective on where we've been and where we're headed.

JVM Language Summit

JULY 28–30
SANTA CLARA, CALIFORNIA

The 2014 JVM Language Summit is an open technical collaboration among language designers, com-

piler writers, tool builders, runtime engineers, and virtual machine (VM) architects.

The Developer's Conference

AUGUST 3–9
SÃO PAULO, BRAZIL

One of Brazil's largest developer conferences, this event offers 35 tracks, several of them exclusively on Java.

JCrete

AUGUST 25–29
CHANIA, GREECE

JCrete is a Java unconference for Java experts and evangelists and Java Champions. It combines the benefits of an open-spaces conference with Greek hospitality and seaside location.

JCertif

SEPTEMBER 8–14
BRAZZAVILLE, CONGO

JCertif is the biggest IT community event in central Africa. International speakers present talks and labs about Java technologies, web apps, cloud apps, and more.

JavaZone

SEPTEMBER 9–11
OSLO, NORWAY

The conference offers a combination of technical talks and panels in an informal atmosphere with an expected attendance of more than 2,500. JavaZone 2014 will be the 13th consecutive JavaZone conference.

JAVA IN ACTION



JAVA TECH

ABOUT US



©

Java
.ne

blo



17



And, of course, people will come to JavaOne for the speakers, including Jim Manico, Hendrik Ebbers, Rebecca Parsons, and David Blevins. Read what these technology experts have to say about their upcoming sessions and what they're looking forward to at the conference. Get their advice on how to make the most of your JavaOne 2014 experience.



JIM MANICO

JAVA TECH

ABOUT US

0

f

ava
net

olog

19

*Don't miss Manico's session,
"Third-Party Java Libraries for Secure
Development."*

Learn more about [tracks](#)
and [register for JavaOne](#).



HENDRIK EBBERS

Java Magazine: What made you decide to write a book?

Java Magazine: What will your javaOne session cover?

ers can use JavaFX in production and in enterprise and business applications. They'll get an overview of the various best practices on solving problems such as client/server communications, asynchronous versus synchronous tasks, MVC/MVP [model-view-controller/model-view-presenter] framework-patterned approaches, and how to design complex dialog window flows.

Java Magazine: What is your advice to developers attending JavaOne? What should they look for, and what should they expect?

Ebbers: There will be some very cool JavaFX, community, open source, and Internet of Things talks this year. The talks are always very professional; I have never experienced a bad one. So don't miss the talks.

Java Magazine: What are you looking forward to at the conference?

Ebbers: I love all things around JavaOne. There are a lot of people whom I see only one or two times a year. JavaOne is the best place to meet with people because everyone is there. The conference has interesting topics. And I like seeing San Francisco and going to parties. All that stuff.

*Don't miss Ebbers' session,
"JavaFX Enterprise."*

JAVA TECH

Get **COFFEE** without leaving the vicinity of the conference. This list of java joints is arranged by walking time from JavaOne at the Hilton Union Square.

333 O'Farrell Street



375 Taylor Street
2-minute walk

Although it offers only plain coffee and no specialty drinks, it also serves a hearty breakfast.

488 Post Street
3-minute walk

55 Cyril Magnin Street
4-minute walk

PHOTOGRAPH BY GREENCOLANDER/CC BY



REBECCA PARSONS

JAVA TECH

A tall glass of iced coffee with a swirl of cream and a cinnamon stick. The glass is filled with a light brown liquid, topped with a thick layer of white cream and a cinnamon stick. The background is a blurred wooden surface.

ABOUT US



DAVID BLEVINS

Java Magazine: You're giving two talks at JavaOne. With one, you appear to be taking on the role of Java EE champion/promoter.

Blevins: Yes. I want to get people up to speed on the major high-level, macro-level changes we've made to realign Java EE to what is

Don't miss Blevins' sessions, "Java EE Game Changers" and "Apache TomEE, Java EE Web Profile and More on Tomcat."

Tom Caldecott is a writer in Oracle Brand Communications.

JAVA TECH

501 Jones Street
Experience a speakeasy
from the Prohibition Era
(make reservations to get
the password).



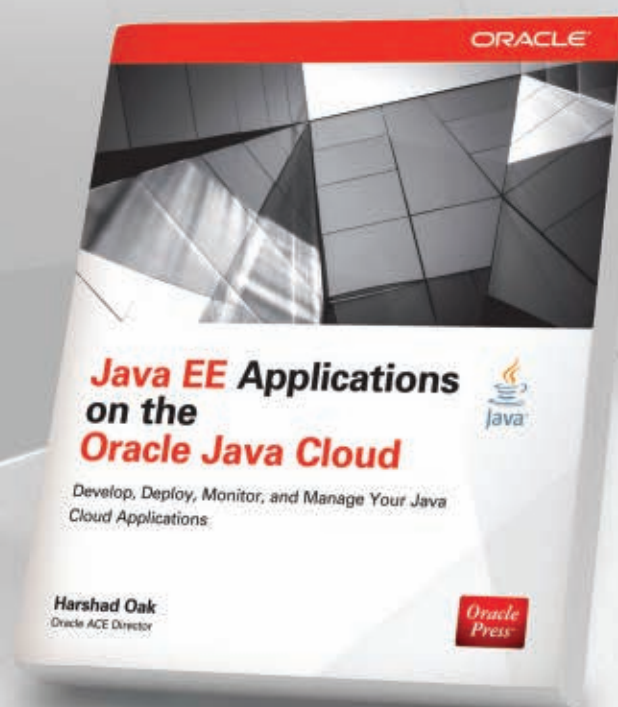
4 Valencia Street
Sip a martini and sing along
with the piano player.

401 Taylor Street
Want a beer? They have
18 on tap.

—Curran Mahowald

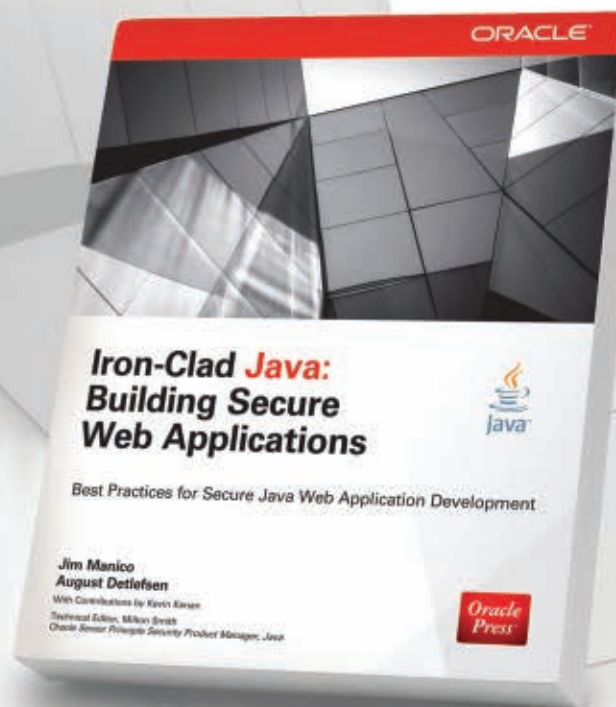
PHOTOGRAPH BY GETTY IMAGES

Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



**Java EE Applications
on the
Oracle Java Cloud**
Harshad Oak

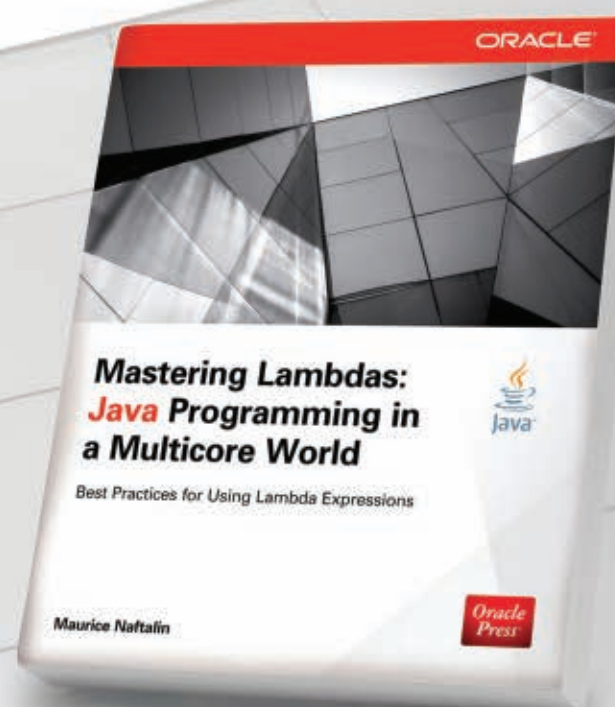
Develop, deploy, and manage
cloud-based Java EE applications



**Iron-Clad Java:
Building Secure Web
Applications**

Jim Manico, August Detlefsen

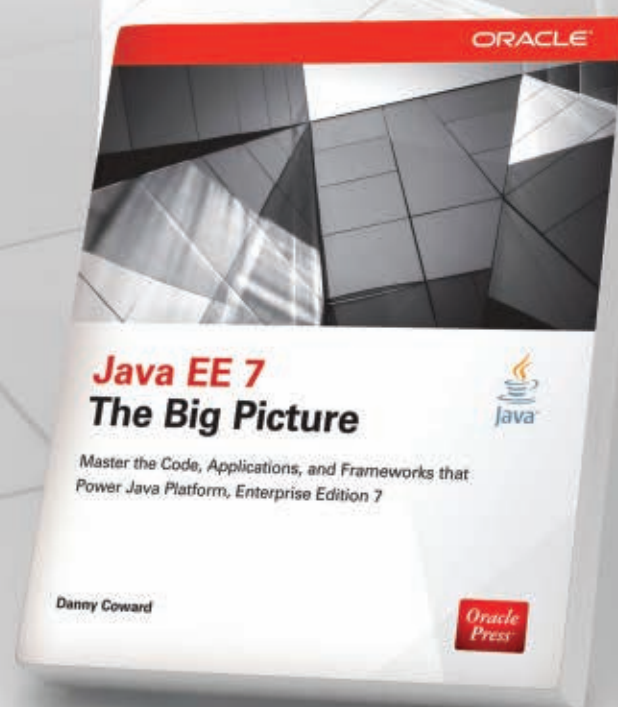
Best practices for developing highly
secure Java-based web applications



**Mastering Lambdas:
Java Programming in
a Multicore World**

Maurice Naftalin

Effectively use Lambda expressions to
maximize performance improvements
provided by multicore hardware



**Java EE 7:
The Big Picture**
Danny Coward

Master the code, applications, and
frameworks that power Java Platform,
Enterprise Edition 7

From left to right:
VNIIRA's Alexandre
Teterin, Anton Fedorov,
and Mikhail Kuznetsov
at the Pulkovo-1
Airport in Saint
Petersburg, Russia

A full-page photograph showing three men walking towards the camera on a paved path. The man on the left wears a plaid shirt and khaki pants. The man in the center wears a light yellow shirt and blue jeans. The man on the right wears a light blue shirt and blue jeans. All three have shoulder bags. In the background, there is a road with cars, green grass, and a clear blue sky with scattered white clouds. A red and white airplane is flying in the upper right corner of the sky.

From left to right:
VNIIRA's Alexandre
Teterin, Anton Fedorov,
and Mikhail Kuznetsov
at the Pulkovo-1
Airport in Saint
Petersburg, Russia

A full-page photograph showing three men walking towards the camera on a paved path. The man on the left wears a plaid shirt and khaki pants. The man in the center wears a light yellow shirt and blue jeans. The man on the right wears a light blue shirt and blue jeans. All three have shoulder bags. In the background, there is a road with cars, green grass, and a clear blue sky with scattered white clouds. A red and white airplane is flying in the upper right corner of the sky.

Fedorov and his team have had plenty of experience with air traffic control systems in their work for the Scientific Research Center of Air Traffic Management, a division of the

vniira-ovd.com

Aerospace

Saint Petersburg,
Russia

1,500

used:

Java SE 1.7,
NetBeans IDE 7.4

A group of young men are gathered around a computer desk in a lab setting. One man in a yellow shirt is pointing at the screen, while another in a plaid shirt is using a mouse. Other students are visible in the background, also looking at the screen.

"The airport is absolutely in the middle of nowhere," explains Fedorov. "The

trollers are the primary users of these systems, which track thousands of daily flights using primary and secondary surveillance radars.

Like most air traffic control systems, SINTEZ handles airspace organization and management, aerodrome

operations, demand and capacity balancing, traffic synchronization, conflict management, and airspace user operations. The airport relies on the custom system to process and display surveillance data and flight data, including information regarding airspace management, meteorological data, and air safety.

“SINTEZ predicts aircraft flight paths over a wide area of interest with great precision, allowing air traffic controllers to detect and forecast potential conflicts,” explains Teterin. The system also helps air traffic controllers resolve these conflicts in an expedient manner. Application features include flight tracking, collision and resolution, and data from various sources, together with logging and recording of all system and user actions.

SINTEZ consists of dozens of software modules to execute and coordinate these functions. Some of these components

are implemented only in Java, others in Java and C++, and others only in C++. Fedorov and his team used Java with the NetBeans IDE because of the modular, cross-platform nature of this system.

"This modular open architecture made it easy to accommodate to shifting needs throughout the development lifecycle, as well as to interface with many types of radar systems and other sources of information," Fedorov says. "We were able to compose features from multiple modules to provide the necessary functionality tailored to various roles, from flight executive officers

to air traffic controllers."

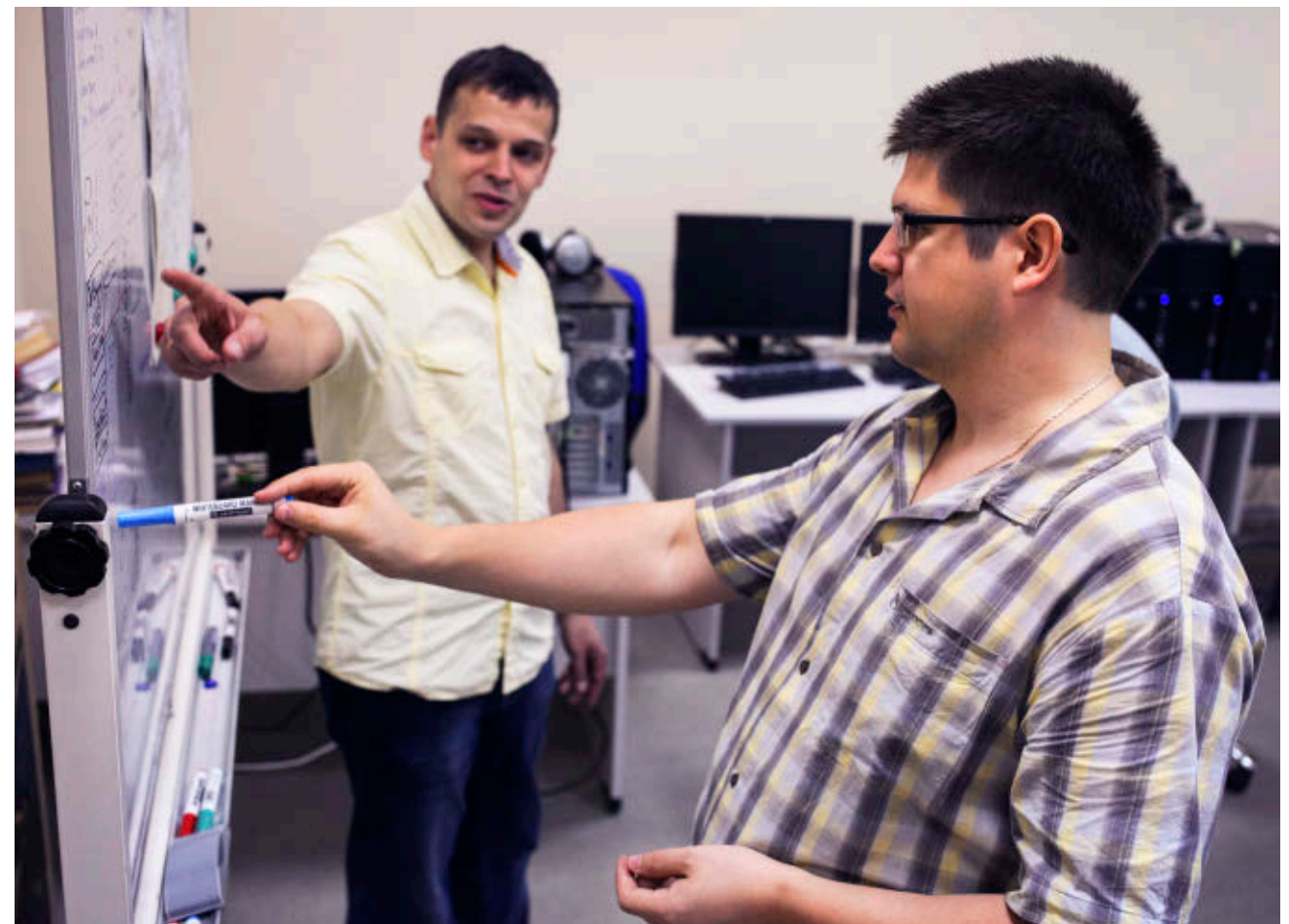
For example, the NetBeans Visual Library permitted the developers to create custom UI controls. According to Teterin, utilizing different sets of modules makes it possible to deploy automated workstations, which ensure execution of various tasks: flight management, air traffic controlling, engineering, and so forth.

The team also favored Java for its rapid application development cycle and innate portability. The NetBeans IDE can run on Windows, Macintosh OS X, Linux, Oracle Solaris, and other

VNIIRA's Credentials

VNIIRA has a great depth of experience developing customized navigation and flight solutions—and a proud list of accomplishments:

- 65 years as a pioneer of air safety
- 150 prototypes of radio technical systems, navigation systems, and radio instruments
- 1,300 inventor certificates and patents
- 60 complexes of air traffic control systems and facilities in Russia and other countries
- 100 types of aircraft employing VNIIRA's airborne equipment, navigation, and landing facilities



Fedorov (left) and Teterin brainstorm about updates to the SINTEZ system.



Interactive Objects with BlueJ

In this second part of a two-part series, we continue our discussion of BlueJ's interaction features—the main aspect that differentiates BlueJ from other environments. In [Part 1](#), we discussed why interaction and visualization are important for learners, and we started by demonstrating the first (and most fundamental) examples of the visualization of object-oriented concepts in BlueJ: the depiction of classes and objects. Both are visualized graphically and allow direct interaction that illustrates and reinforces their roles, characteristics, and behavior.

design elements and tools that help learners develop consistent mental models and explore aspects of object orientation.

When learners start working in BlueJ, they can work at a conceptual level that is abstracted from some of the underlying implementation details. For example, users of BlueJ create *classes*, not *files*. The distinction might be subtle at first, but it is meaningful. Users do not need to be concerned with the file system structure, or, for example, with the rule that the filename has to match the class name. In fact, they do not even need to know, initially, that

changes the package name,
BlueJ moves the source file to
the right location.

At this point, experienced programmers often say "But it is important to know how Java classes are stored," to which my answer is, "No, it



a result, the result is displayed in a separate dialog box. The user can then inspect the result or—if it is an object—place it on the object bench. If the user is in the process of recording a unit test, an assertion can also be added.

Taken together, interactive method invocation, parameter passing, and return value display provide a consistent and meaningful illustration of object communication in object-oriented systems. Performing this process interactively imbues students with a sense of how this interaction works, and it helps them gain a good understanding of the mechanism.

Inspection

Methods, however, do not always return values; their effect might be a state change rather than an explicit result. *State* is the third of the three characteristics that define objects in object-oriented systems. (The other two are *identity* and *behavior*—and we have already seen how these are illustrated through interactive object creation, object display on the object bench, and method invocation.)

State is also visualized in BlueJ. Each object's context menu includes an **Inspect** option, which displays a visualization of the object's internal state (see **Figure 3**). Fields of the objects are

listed with their type and name, and current values are displayed.

In the early phases of learning, it is especially enlightening to inspect two objects of the same type side by side and observe the similarity in fields, but the differences in values. Object inspectors can also remain open during interactive method calls to observe a state change more immediately.

Again, this tool visualizes an important principle of object-oriented programming and supports obtaining a valid model of machine behavior.

Static methods and inspection.

Static methods can also be invoked interactively. This is done by selecting them from the class'—rather than the object's—context menu. Similarly, static fields are displayed by inspecting the class rather than the object. This distinction in the interface

reinforces the differences in definition between static attributes and object attributes.

The Editor: Scope Highlighting

As every teacher knows, understanding the object model is not the only difficulty in learning to program in Java. Beginning students also struggle with details of Java's syntax when writing their programs.

One of the most common problems, especially for young learners, is the correct balancing of curly

brackets to define scopes. The very concept of nested scopes—so seemingly natural for all of us who have programmed for some time—is difficult to grasp for many beginners, and the placement of matching opening and closing scope brackets is prone to errors. Professional environments have tried to help by automating the insertion of bracket pairs and by highlighting matching brackets. However, this has not caused a great reduction in beginners' errors.

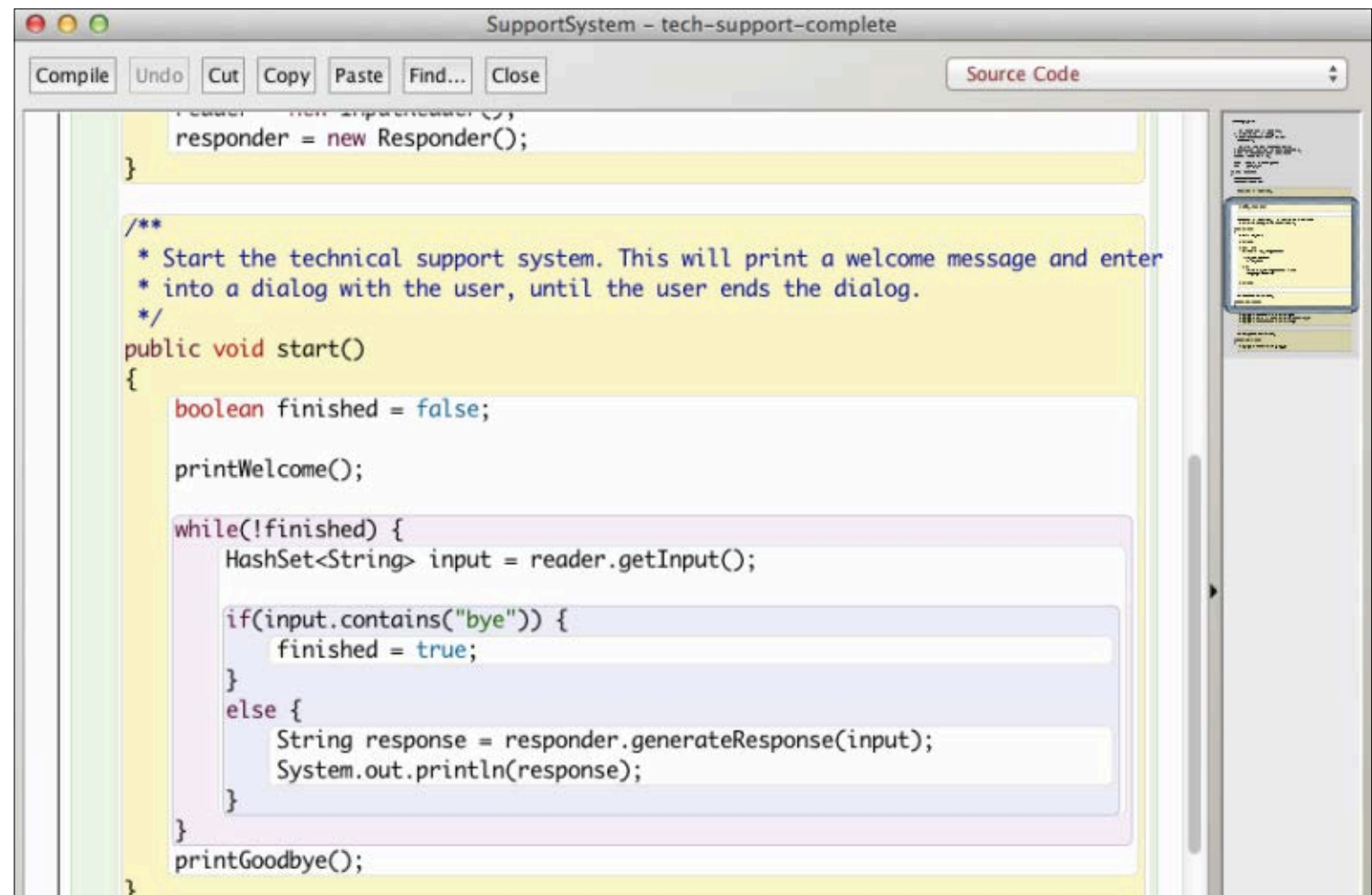


Figure 4



LAW IN ACTION

JAVA TECH

the **PATH** environment variable is set up appropriately.

Quickly Reading a File Using Groovy and Scala

In **Listing 1**, we first open the `/etc/networks` file using the `BufferedReader` class (replace `/etc/networks` with a filename that is valid for your system). Then we loop through each line to concatenate the content into a `StringBuilder`. The noise in the code comes from two parts: the lack of a convenience method in `BufferedReader` and the need to handle checked exceptions. We can reduce the noise just a bit by using the `BufferedReader`'s `lines()` method in Java 8; however, the noise from checked

Leveraging the Polyglot Ecosystem



Java
Virtual
Machine

exceptions is here to stay.

We can quickly read the entire contents of a relatively small file using one line of code in both Groovy and Scala. Let's take a look at an example in Groovy first.

In **Listing 2**, `File` is the all-too-familiar class from the JDK `java.io` package. Groovy has extended this class with a `getText()` method, which we can access using the `text` property. Furthermore, Groovy does not have checked exceptions, so we're not forced to handle any exceptions. (Any exception we don't handle will propagate much like unchecked exceptions do in Java.)

To run the code shown in **Listing 2**, type the following command on the command line, where `readFile.groovy` is the file in which the Groovy code is saved:

■ groovy readFile.groovy

In case we mistype the filename, we can place a `try` and `catch` block around the code, if we desire, to handle any exceptions such as “file not found.” Replace the constructor argument `/etc/networks` with other filenames to get the contents of different files.

Let's take a look at similar code in Scala. In **Listing 3**, the `fromFile()` method of the `Source` class can read the entire content of a given file and return an iterator. The `mkString`

method concatenates the lines into a single string. Instead of using this method, we could iterate over each line, processing one line at a time.

To run the code in **Listing 3**, type the following command on the command line, where `readFile.scala` is the file in which the Scala code is saved:

scala readFile.scala

Easily Starting a Process Using Groovy

On a project I was working on, I was able to replace 50 lines of code for starting and interacting with an external program with a mere three lines of Groovy code, thanks to some convenience methods in Groovy.

In the previous section, we saw how Groovy took the `File` class and extended it with a convenience method. Likewise, Groovy adds some convenience methods to the `java.lang.Process` class, which we can use to easily start external processes.

Let's take a look at an example to call an external program. To keep the output short, we will simply call `git` but ask only for the version number. **Listing 4** shows the code.

To run an external process, we first create a string of the command we want to run, such as `git version` in this example. Then we call the

LISTING 1

LISTING 2

LISTING 3

LISTING 4

```
//]java
import java.io.*;

public class ReadFile {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(
                new FileReader("/etc/networks"));

            StringBuilder fileContent = new StringBuilder();
            String line;
            while((line = reader.readLine()) != null) {
                fileContent.append(String.format("%s\n", line));
            }

            //use fileContent for something practical here...
            System.out.println(fileContent);
        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```



Download all listings in this issue as text

`execute()` method on it. This is a Groovy-created extension method on the `java.lang.String` class. This method returns a `java.lang.Process` instance on which we call the `getText()` method (using the `text` property). This method returns the standard output of the program we invoked.

To run the program, save it in a file named `callGit.groovy` and type

the following command on the command line:

```
groovy callGit.groovy
```

Here's the output from the code:

git version 1.9.2

In this example, we merely printed the content from the stan-

the end. This simple task seems to be a slippery slope. Let's take yet another stab at it, this time to get rid of that trailing comma (see **Listing 7**).

■ Marlin, Gill, Greg, Deb, Bruce

Thankfully, the way we do this task has changed in Java SE 8. The `String` class has a new `join` method that can save us from all this trouble. **Listing 8** makes use of this new method.

With one call to the `join` method, we instructed Java to concatenate the elements in the list into a single string. We went from the verbose version that had a primitive obsession to a declarative version in which we concisely specified our intention. If we want to perform some operations on the list of elements and then concatenate the result, we can use the `joining` method of the `Collectors` utility class with the `collect` method of the `Stream` interface.

Joining Values Using Java SE 8

Let's compile and run the code in **Listing 5** from the command line:

```
javac PrintList.java
java PrintList
```

Here is the output from the code:

■ [Marlin, Gill, Greg, Deb, Bruce]

It's not quite what we wanted; the names are comma-separated, but we have to get rid of those square brackets. Let's try again using the code in **Listing 6**, which uses the Java SE 5 [for-each](#) construct to loop through the values and print them in comma-separated form. Now we get this output:

Marlin, Gill, Greg, Deb, Bruce,

We got rid of the square brackets, but there is a silly comma at

LISTING 5 / LISTING 6 / LISTING 7 / LISTING 8 / LISTING 9 / LISTING 10

```
//)java
import java.util.*;

public class PrintList {
    public static void main(String[] args) {
        List<String> names = Arrays.asList(
            "Marlin", "Gill", "Greg", "Deb", "Bruce");

        System.out.println(names);
    }
}
```



Download all listings in this issue as text

to the classes and interfaces in the JDK to make our lives easier. Take a few minutes to revisit the JDK classes and interfaces to see what's new in Java SE 8.

Using Delegation in Groovy

Inheritance is quite useful for substitutability when an instance of a class can be used in place of an instance of a base class. However, if we desire class reuse more than substitutability, it is better to use delegation instead of inheritance.

Yet, in Java we often use inheritance more than delegation. The fact that delegation is difficult to write while inheritance is easy to implement is one of the reasons. If delegation were just as easy, we might rely on it more.

Using compile-time metaprogramming, Groovy makes delega-

tion quite easy to use. Let's look at an example. The two classes in **Listing 9** are written in Groovy; however, they could be written in any JVM language.

The `Worker` class has one method, `work`, which prints out a message when called. The `Analyst` class has two methods, one with the same name and signature as the method in the `Worker` class.

Let's now create a Groovy class named `Boss`, which—as you might expect—does nothing. However, this is one smart class; it knows how to tactfully delegate (see **Listing 10**).

Within the `Boss` class, we first use the `@Delegate` annotation on an instance of the `Worker` class. This tells the Groovy compiler to generate in the `Boss` class a method named `work` that merely routes the


```
//Groovy
def boss = new Boss()
boss.work()
boss.analyze()
```

We created an instance of **Boss** and invoked the methods **work** and **analyze**. Before we run this code, let's take a look at the compiler magic. Save the **Worker** class, the **Analyst** class, and the **Boss** class plus the code above for using the **Boss** class in a file named **delegate.groovy**. Then, as shown below, compile the file using the **groovyc** compiler, which will produce Java bytecode for the three classes:

```
groovyc delegate.groovy
```

Let's take a peek at the bytecode for the `Boss` class using the `javap` tool:

```
javap -c Boss.class
```

If you examine the output of this tool, you will find that the compiled `Boss` class contains a method named `work` where the implementation routes the call to the `worker` instance's `work` method. See the excerpt shown in **Listing 11**.

Likewise, the `analyze` method will route the call to the `analyze` method of the `analyzer` instance.

Run the `delegate.groovy` file from the command line:

groovy delegation.groovy

Here is the output from the code:

working...
analyst analyzing...

Groovy has a number of annotations such as `@Delegate` that perform compile-time abstract syntax tree (AST) transformations. Since these are done at compile time, there is no overhead at runtime for using these features. So, they provide convenience without compromising runtime performance.

Performing Pattern Matching in Scala

There's often a need to parse the content of a file or content received through a messaging service. The pattern matching capability of Scala is one of my favorite features of the language. It provides a concise

LISTING 12

```
Compiled from "delegate.groovy"
public class Boss implements groovy.lang.GroovyObject {

...
    public void work();
    Code:
        0: aload_0
        1: getfield    #29                // Field worker:LWorker;
        4: invokevirtual #45              // Method Worker.work():V
        7: aconst_null
        8: pop
        9: return
...
}
```

 [Download all listings in this issue as text](#)

way to not only take actions based on the types of data, but also to extract the contents from select types. Let's look at the example shown in **Listing 12**.

The `process` method receives an object (`Any` in Scala is like `Object` in Java). We invoke the `match` method on the instance and provide it several `case` statements to match against. In the first `case`, we look for an instance of `String`. If that match does not succeed, Scala will continue looking further in the sequence. We then look for the literal `5` followed by any instance of `Int`.

In the next `case`, we look for a list that starts with two elements: `apple` and `peach`. We capture the remaining elements in a `fruits` variable and print it comma-separated using the `mkString()` method (this method is the Scala equivalent of the Java SE 8 `joining` method we discussed earlier).

In the final `case`, we look for an XML fragment and parse the child of the `greet` element. In each of the `case` matches, we print a message with details about the parsed content.

As an aside, in **Listing 12** we see how Scala treats XML as a first-

class citizen. We can place any well-formed XML directly into Scala code, and we can embed Scala expressions into XML for easy generation of XML documents.

Let's call the `process` method with a few different values, as shown in **Listing 13**.

Now, let's look at the output first and then discuss each of the calls. To run the code, save the code in **Listing 12** and **Listing 13** in a file named `match.scala` and then type the following on the command line:

scala match.scala

Listing 14 shows the output.

The call to `process` with `hello` as an argument was picked up by the first `case` since there was a direct type match. The second call with a value of `5` was matched by the literal in the second `case`. The argument of `2`, on the other hand, was matched by the `case` with the `Int` type. The `List` argument was matched by the `case` with the list, and the `fruit` variable was bound to the values `grapes` and `kiwi`.

Further, the XML fragment we passed in was matched by the last `case` and the child of the `greet` element, `howdy`, was bound to the `msg` variable. The last call with argument `2.2` was unmatched since no `case` exists for that value or that type. Therefore, the last call generated

a runtime `MatchError` exception,
which we handled on the call side.

In applications where we want to match against different types, values, or both, we can make use of the Scala matching capability to succinctly compare and process the values.

Easily Parsing XML Using Groovy

Groovy's [XmlSlurper](#), combined with the dynamic nature of the language, makes parsing XML documents effortless. Let's get a feel for its capabilities using an example XML document, [languages.xml](#), which is shown in **Listing 15**.

The root element, `languages`, has a bunch of child elements named `language`. Each of the child elements contains a `name` attribute and an `author` child element. Let's use the code in **Listing 16** to parse this XML content and print the name and author of each of the languages mentioned in `languages.xml`.

Listing 16 creates an instance of the `XmlSlurper` and points its `parse` method at the `languages.xml` file. The parser creates an instance of a node representation. We can dynamically access the properties of this object using the names of the known elements in `languages.xml`. To access an attribute, we prefix the name of the attribute with the `@` symbol, for example, `@name`.

LISTING 13

LISTING 14

LISTING 15

LISTING 16

LISTING 17

```
//Scala
process("hello")
process(5)
process(2)
process(List("apple", "peach", "grapes", "kiwi"))
process(<greet>howdy</greet>)
try {
  process(2.2)
} catch {
  case ex : MatchError => println(s"Error: $ex")
}
```

 [Download all listings in this issue as text](#)

Let's run this code and take a look at the output it produces:

Languages and authors:

C++	Stroustrup
Java	Gosling
Scala	Odersky
Ruby	Matz
Lisp	McCarthy

In addition to simple iteration over the elements, we can also apply powerful filter and search operations to extract the desired contents or parts of the document.

Generating XML Documents Using Groovy

In the previous section, we saw how to parse an XML document using Groovy. Groovy also makes it quite easy to create an XML document from data in memory (or data read from a file or from a database). Let's create a hashmap with sample data for languages and authors, and then create code to produce the `languages.xml` file we used earlier.

In **Listing 17**, the `langs` reference holds a hashmap of some

In addition to Groovy, we could also fluently create XML documents using Scala, since that language treats XML as a first-class citizen and allows you to embed expressions in XML content.

As a final step, we're ready to use the facility we've created. Instead

```
//Groovy
import java.time.*

class DateUtil {
    int number

    DateUtil(aNumber) { number = aNumber }

    def getAgo() {
        LocalDateTime.now().minusDays(number)
    }
}

Integer.metaClass.getDays = { -> return new DateUtil(delegate) }

println 2.days.ago
```



Using a recursive approach, we can solve some problems by using solutions to their subproblems. Let's first take a look at the small Scala example shown in **Listing 19**, and discuss the problems we will soon run into with this approach.

This problem can be avoided using a special technique called *tail-call optimization*. A tail-call is

multiply the partial result in **fact** with the current **number** and pass it as an argument to the **factorial** method. This recursive call is the last operation in the method when the recursion has to continue.

At compile time, the `tailrec` annotation will ensure that the method is in tail-recursive form. The compiler will optimize this code “under the hood” by converting the recursion into a mere iteration. Even for large input parameters, this code will not run into a stack overflow situation.

With this approach in mind,
when working on applications
where we want to employ recur-

sive algorithms for large input sizes, we can benefit from tail-call optimization techniques.

Guaranteeing Resource Cleanup in Java

With the introduction of lambda expressions in Java, we can make use in Java of some good old patterns from the Smalltalk days. One such pattern is the Execute Around Method pattern, which is useful for wrapping a piece of code around some logic that we want to ensure

In this new version, the `factorial` method takes two parameters: the partial factorial result in the `fact` parameter and the `number` for which the factorial needs to be computed. Unlike the code in **Listing 19**, the last operation in **Listing 20** is not multiplication. We

GO EXPLORING

By looking beyond the syntax—**by exploring the libraries and key capabilities of the languages we adopt**—we can leverage the languages to perform various routine tasks with greater ease.

LISTING 20

LISTING 21

LISTING 22

```
//Scala
@scala.annotation.tailrec
def factorial(fact: BigInt, number: Int) : BigInt = {
  if(number == 1)
    fact
  else
    factorial(fact * number, number - 1)
}
```



Download all listings in this issue as text

is run. Let's first discuss the problem and then look into why this pattern is a good option.

Suppose we have an object that uses extensive external resources and we want to clean up the object deterministically. Java SE 7 provides a special language feature called **try-with-resources**. This feature is also called Automatic Resource Management (ARM). Let's look at an example of this feature, discuss some problems with it, and then look at an alternative solution.

Listing 21 shows a `ResourceARM` class that uses the ARM feature in Java SE 7.

To use the `try-with-resources` feature on a class, the class should implement the `AutoCloseable` interface. The `ResourceARM` class implements that interface and the required `close` method that would actively clean up the resources.

Let's use this class in an example.

In **Listing 22**, we placed the instance creation in the special form of `try`. This form removes quite a bit of noise in code; the `finally` block is not required because that part is synthesized by the compiler. In the synthesized `finally` block, the compiler automatically calls the `close` method on the instance we created in the `try` statement.

With ARM, we have two benefits: the code is reduced—since we don't have to write the `finally` block—and the resource cleanup is guaranteed at the end of the `try` block. This `try`-with-resources feature is useful when such deterministic cleanup is necessary.

The disadvantage of this feature, however, is that we have to use this form of the `try` block. ARM is a suggestive feature; it does not force us to do the right thing.

we don't have to create an instance of single, abstract method interfaces; we can use lambda expressions instead. Here is the code to use [Resource](#):

```
//java
Resource.use(resource -> {
    resource.op1();
    resource.op2();
});
```

Since we marked the constructor `private`, the use of the pattern is forced on programmers using the `Resource` class. Unlike the ARM feature, the Execute Around Method pattern is useful when the pre- and post-operations have to be guaranteed and the usage of the instance has to be rather narrow and enforced.

Simple tasks should be simple,
and thanks to some highly capable

```
//Java
import java.util.function.Consumer;

public class Resource {
    private Resource() { System.out.println("Resource created..."); }
    public void op1() { System.out.println("op1"); }
    public void op2() { System.out.println("op2"); }
    private void close() { System.out.println("cleanup logic goes here..."); }

    public static void use(Consumer<Resource> block) {
        Resource resource = new Resource();
        try {
            block.accept(resource);
        } finally {
            resource.close();
        }
    }
}
```

 [Download all listings in this issue as text](#)

By looking beyond the syntax—by exploring the libraries and key capabilities of the languages we adopt—we can leverage the languages to perform various routine tasks with greater ease. **</article>**

MORE ON TOPIC:



- Java SE 8
- Groovy
- Scala

EASY DOES IT
Tasks that are arduous in one language **might be quite easy** to perform in another.



Use JMH to write useful benchmarks that produce accurate results.

Yet, evaluating the performance of certain critical pieces of code is essential for developers who create applications, frameworks, and tools. Stressing critical portions of code and obtaining metrics that are meaningful is actually difficult in the Java Virtual Machine (JVM) world, because the JVM is an adaptive virtual machine. As we will see in this article, the JVM

In this article, we will start by creating a simple yet naive benchmarking framework. We will see why things do not turn out as well as we hoped. We then will look at JMH, a benchmark harness that gives us a solid foundation for writing benchmarks. Finally, we'll discuss how JMH makes writing concurrent benchmarks simple.

Benchmarking does not seem so difficult. After all, it *should* boil down to measur-

The `bench` method executes a benchmark expressed as a `java.lang Runnable`. The other parameters include a descriptive name (`name`), a benchmark run duration (`runMillis`), the inner loop upper bound (`loop`), the number of warm-up rounds (`warmup`), and the number of measured rounds (`repeat`).

Sample usage. Let's use our fresh benchmarking framework with the following method:

The `distance` method computes the Euclidean distance between two points (`x1`, `y1`)

This is the first lesson: **mixing benchmarks** within the same JVM run is wrong.



PHOTOGRAPH BY
MATT BOSTOCK/GETTY IMAGES

Let's introduce the following constants for our experiments: 4-second runs, 10 measurements, 15 warm-up rounds, and an inner loop of 10,000 iterations:

Running the benchmark is done as follows:

On a test machine, a random execution produces the following shortened trace:

According to our benchmark, the **distance** method has a throughput of 30483613 operations per millisecond (ms). Another run would yield a slightly different throughput. Java devel-

Great; but still . . . is 30483613 operations per ms for **distance** a meaningful result?

The raw throughput value does not give us much perspective, so let's compare our result for **distance** with the throughput of other methods.

Looking for a baseline. Let's take the same method signature as `distance` and return a constant instead of doing a computation with the parameters:

We also update our benchmark as shown in **Listing 2**. The `constant` method will give us a good baseline for our measurements, since it just returns a constant. Unfortunately, the results are not what we would intuitively expect:

LISTING 2

 [Download all listings in this issue as text](#)

tries to speculate on the entire code using optimizations such as loop unrolling, method inlining, and on-stack replacements.

Calling `System.currentTimeMillis()` has an effect on throughput as well, and our benchmarks would need to accurately subtract the time taken for each of these calls. We could also play with different inner-loop upper-bound values and observe very different results.

The [OpenJDK wiki Performance Techniques](#) page provides a great overview of the various techniques being used in Java HotSpot VM. As you can see, ensuring that we measure only the code to be benchmarked is difficult.

More pitfalls. Going back to the performance evaluation of the `distance` method, we noted that its throughput was very similar to the throughput measured for a method that would do no computation and return a constant.

In fact, Java HotSpot VM used dead-code elimination; since the return value of `distance` is never used by our `java.lang.Runnable` under test, it practically removed it. This also happened because the method has no side effect and has a simple control flow that is recursion-free.

To convince ourselves of this, let's modify the `java.lang Runnable` lambda that we pass to our benchmark method, as shown in **Listing 5**.

Instead of just calling `distance`, we now assign its return value to a field and eventually print it, to force the virtual machine not to ignore it. The benchmark figures are now quite different:

```
Running: distance_use_return
(...)
[ ~18865939 ops/ms ]
```

We now have a more meaningful result, because the `constant` method had a throughput of about 30229883 operations per ms on our test machine.

Although it's not perceptible in this example, we could also highlight the effect of *constant folding*. Given a simple method with constant arguments and a return value that is evidently dependent on those parameters, the virtual machine is able to speculate that it is not useful to evaluate each call. We could come up with an example to illustrate that, but let's instead focus on writing benchmarks with a good harness framework.

Indeed, the following should be clear by now:

- Our simple benchmarking framework has flaws.
- The virtual machine does so many optimizations that it is difficult to ensure that what we are benchmarking is actually what we expect to benchmark.

LISTING 5 LISTING 6 LISTING 7

```
// (...)

static double last = 0.0d;

public static void main(String... args) {
    bench("distance_use_return", RUN_MILLIS, LOOP, WARMUP,
    REPEAT, () -> last = distance(0.0, 0.0, 10.0, 10.0));
    System.out.println(last);
}
```

 [Download all listings in this issue as text](#)

Introducing JMH

JMH is a Java harness library for writing benchmarks on the JVM, and it was developed as part of the OpenJDK project. JMH provides a very solid foundation for writing and running benchmarks whose results are not erroneous due to unwanted virtual machine optimizations. JMH itself does not prevent the pitfalls that we exposed earlier, but it greatly helps in mitigating them.

JMH is popular for writing *micro-benchmarks*, that is, benchmarks that stress a very specific piece of code. JMH also excels at concurrent benchmarks. That being said, JMH is a general-purpose benchmarking harness. It is useful for larger benchmarks, too.

Creating and running a JMH project.

While JMH releases are being regularly published to Maven Central Repository, JMH develop-

ment is very active and it is a great idea to make builds of JMH yourself. To do so, you need to clone the JMH Mercurial repository, and then build it with Apache Maven, as shown in **Listing 6**. Once this is done, you can bootstrap a new Maven-based JMH project, as shown in **Listing 7**.

This creates a project in the **benchmarks** folder. A sample benchmark can be found in `src/main/java/MyBenchmark.java`. While we will dissect the sample benchmark in a minute, we can already build the project with Apache Maven:

```
$ cd benchmarks/  
$ mvn package  
(...)  
$ java \  
-jar target/microbenchmarks.jar  
(...)
```

When you run the self-contained `microbenchmarks.jar` executable JAR file, JMH launches all the benchmarks of the project with default settings. In this case, it runs `MyBenchmark` with the default JDK and no specific JVM tuning. Each benchmark is run with 20 warm-up rounds of 1 second each and then with 20 measurement rounds of 1 second each. Also, JMH launches a new JVM 10 times for running each benchmark.

As we will see later, this behavior can be customized in the benchmark source code, and it can be overridden using command-line flags. Running `java -jar target/microbenchmarks.jar -help` allows us to see the available flags.

Let's instead run the benchmark with the parameters shown in **Listing 8**. These parameters specify the following:

- We use only one fork (**-f1**).
- We run five warm-up iterations (**-wi 5**).
- We run five iterations of 3 seconds each (**-i 5 -r 3s**).
- We tune the JVM configuration with **jvmArgs**.
- We run all benchmarks whose class name matches the **.*Benchmark.*** regular expression.

The execution gives a recap of the configuration, the information for each iteration and, finally, a summary of the results that

includes confidence intervals, as shown in **Listing 9**.

Anatomy of a JMH benchmark.

The sample benchmark that was generated looks like **Listing 10**. A JMH benchmark is simply a class in which each `@GenerateMicroBenchmark` annotated method is a benchmark. Let's transform the benchmark to measure the cost of adding two integers (see **Listing 11**).

We have a **baseline** benchmark that gives us a reference on returning an **int** value. JMH takes care of reusing return values so as to defeat dead-code elimination. We also return the value of field **x**; because the value can be changed from a large number of sources, the virtual machine is unlikely to attempt constant folding optimizations. The code of **sum** is very similar.

The benchmark has more configuration annotations present. The `@State` annotation is useful in the context of concurrent benchmarks. In our case, we simply hint to JMH that `x` and `y` are thread-scoped. The other annotations are self-explanatory. Note that these values can be overridden from the command line. By running the benchmark on a sample machine, we get the results shown in **Listing 12**.

Lifecycle and parameter injection.

In simple cases, class fields can hold the benchmark state values.

LISTING 8 / LISTING 9 / LISTING 10 / LISTING 11 / LISTING 12

```
$ java -jar target/microbenchmarks.jar \
  -f1 -wi 5 -i 5 -r 3s \
  -jvmArgs '-server -XX:+AggressiveOpts' \
  .*Benchmark.*
```

 [Download all listings in this issue as text](#)

The class shown in **Listings 13a** and **13b** is an elaborated version of the previous benchmark with a state class, a lifecycle for the state class, and a **Blackhole**.

JMH was designed with concurrent benchmarks in mind. These kinds of benchmarks are very difficult to measure correctly, because they involve several threads and inherently nondeterministic behaviors.

We are using two baselines. The first is a `void` empty method, and the second simply returns a con-

```
package com.mycompany;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.logic.BlackHole;

import java.util.Random;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@Fork(value = 3,
    jvmArgsAppend = {"-server", "-disablesystemassertions"})
public class MyBenchmark {

    @State(Scope.Thread)
    static public class AdditionState {

        int x;
        int y;

        @Setup(Level.Iteration)
        public void prepare() {
            Random random = new Random();
            x = random.nextInt();
            y = random.nextInt();
        }

        @TearDown(Level.Iteration)
        public void shutdown() {
            x = y = 0; // useless in this benchmark...
        }
    }
}
```



Download all listings in this issue as text

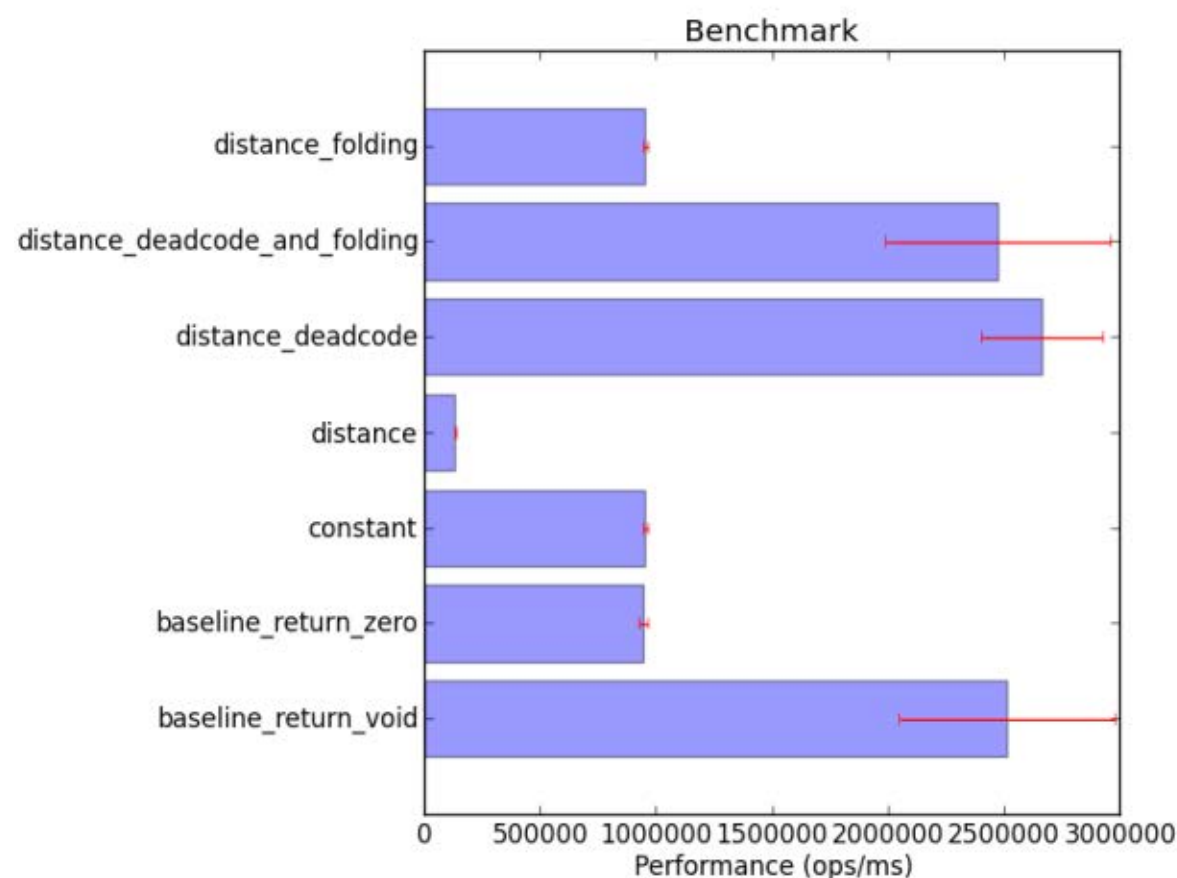


Figure 1

stant `double` value, as shown in **Listing 16**. Benchmarking `constant()` and `distance()` is as simple as **Listing 17**.

To put things into perspective, we also include flawed measurements subject to dead-code elimination and constant folding optimizations (see **Listing 18**).

Finally, we can also provide a `main` method to this benchmark using the JMH builder API, which mimics the command-line arguments that can be given to the self-contained JAR executable. See **Listing 19**.

Figure 1 shows the results as a bar chart with the mean error included for each benchmark.

Given the two baselines, we clearly see the effects of dead-code elimination and constant folding. The only meaningful measurement of `distance()` is when the value is being consumed by JMH and parameters are passed through field values. All other cases converge to either the performance of returning a constant `double` or an empty `void`-returning method.

LISTING 16

LISTING 17

LISTING 18

LISTING 19

```
@GenerateMicroBenchmark
public void baseline_return_void() {

}

@GenerateMicroBenchmark
public double baseline_return_zero() {
    return 0.0;
}
```

 [Download all listings in this issue as text](#)

Devising Concurrent Benchmarks

JMH was designed with concurrent benchmarks in mind. These kinds of benchmarks are very difficult to measure correctly, because they involve several threads and inherently nondeterministic behaviors.

Next, let's examine concurrent benchmarking with JMH for the comparison of readers and writers over an incrementing `long` value. To do so, we use a pessimistic implementation based on a `long` value for which every access is protected by a synchronized block, and an optimistic implementation based on `java.util.concurrent.atomic.AtomicLong`. We want to compare

the performance of each implementation depending on the proportion of readers and writers that we have.

JMH has the ability to execute a group of threads with different benchmark code. We can specify how many threads will be allocated to a certain benchmark method. In our case, we will have cases with more readers than writers and, conversely, cases with more writers than readers.

Benchmarking the pessimistic implementation. We start with the following benchmark class code:

```
@BenchmarkMode(
    Mode.AverageTime)
```


Figure 2

JAVA IN ACTION

JAVA TECH

ABOUT US

○

f


```
public static void main(String... args) throws RunnerException {
    Options opts = new OptionsBuilder()
        .include("*.ConcurrentBench.*")
        .warmupIterations(5)
        .measurementIterations(5)
        .measurementTime(TimeValue.milliseconds(5000))
        .forks(3)
        .result("results.csv")
        .resultFormat(ResultFormatType.CSV)
        .build();
    new Runner(opts).run();
}
```

marking code and quickly realized that the JVM was doing optimizations that rendered the results meaningless. By contrast, JMH provides a coherent framework to write benchmark code and avoid common pitfalls.

tion. Nevertheless, such benchmarks are great quality assets for performance-critical code, and JMH provides a reliable foundation for writing them correctly. [./article>](#)

 Java Virtual Machine

- [OpenJDK Code Tools: JMH](#)
- [JMH samples](#)



blo





Understanding Java JIT Compilation with JITWatch

LAW IN ACTION

This article provides a basic primer on JIT compilation as it happens in Java HotSpot VM. We'll discuss how to switch on simple logging for the JIT compiler and some of the most common (and important) JIT

Let's kick off with a few fundamentals about JIT compilation as it is done in Java HotSpot VM.

Java HotSpot VM automatically monitors which methods are being executed. Once a method has become eligible (by meeting some criteria, such as being called often), it is scheduled for compilation into machine code, and it is then known as a *hot method*. The compilation into machine code happens on a separate

To learn more about the JIT compilation process, see ["Understanding the Java HotSpot VM Code Cache,"](#) and ["Introduction to JIT Compilation in Java HotSpot VM."](#)

The first step to understanding how JIT compilation in Java HotSpot VM is affecting your code is to see which of your methods are getting compiled. Fortunately this is very easy to do, and only requires you to add the `-XX:+PrintCompilation` flag to the script you use to start your Java processes.

A new open source tool called JITWatch is being developed to give developers much better insight into how the JIT compiler treats their code.



PHOTOGRAPH BY
JOHN BLYTHE

JAVA TECH

ABOUT US



Java
.net

blog



Note: The resulting log of compilation events will end up in the standard log (that is, the standard output), and there is currently no way to redirect the entries to another file.

The exact format of the `PrintCompilation` flag's log entries varies between different Java versions. Here are some examples of log formats from different Java versions. **Listing 1** shows an example log from JDK 6.

In the JDK 6 form of the **PrintCompilation** flag's log entries, the first number corresponds to the compilation ID. This ID essentially tracks an individual method as it is compiled, optimized, and possibly deoptimized again.

Thereafter follow some flags that indicate properties of the method; for example, the **s** indicates the method is synchronized, and the **!** indicates the method has exception handlers.

Next comes the name of the method—in fully qualified form—followed by the number of bytes of byte-code contained in the method being compiled. There is a minor

annoyance: the method signatures are not printed out in the output.

Listing 2 shows an example log from JDK 7 onward. The big change in the JDK 7 form of the logs is that the first column is now the time—in milliseconds since the JVM started—at which the compilation occurred. Otherwise, the other fields are essentially the same as with JDK 6.

There are a number of excellent posts on the subject of reading [PrintCompilation](#) output, for example, those by [Stephen Colebourne](#) and [Chris Vest](#), which are both highly recommended.

Now it's time to look at the Java compiler. If you're using a different JVM language, such as Scala or Groovy, then you should be aware that those languages' compilers might alter (mangle) the names of methods and add or remove methods as part of their process for producing class files.

Some JIT Compilation Techniques

One of the most common JIT compilation techniques used by Java HotSpot VM is *inlining*, which is the practice of substituting the body of a method into the

GET TO KNOW IT

The JIT compiler is one of the most important parts of Java HotSpot VM, and yet many Java developers do not know much about it or how to check that their applications work well with the JIT compiler.

LISTING 1

LISTING 2

JDK 6:

```
22  java.util.HashMap::getEntry (79 bytes)
23  s!  sun.misc.URLClassPath::getLoader (136 bytes)
```



Download all listings in this issue as text

places where that method is called. Inlining saves the cost of calling the method; no new stack frames need to be created. By default, Java HotSpot VM will try to inline methods that contain less than 35 bytes of JVM bytecode.

Another common optimization that Java HotSpot VM makes is *monomorphic dispatch*, which relies on the observed fact that, usually, there aren't paths through a method that cause an object reference to be of one type most of the time but of another type at other times.

You might think that having different types via different code paths would be ruled out by Java's static typing, but remember that an instance of a subtype is always a valid instance of a supertype (this principle is known as the *Liskov substitution principle*, after Barbara Liskov). This situation means that there could be two paths into a method—for example, one that passes an instance of a supertype and one that passes an instance of

a subtype—which would be legal by the rules of Java’s static typing (and does occur in practice).

In the usual case (the monomorphic case), however, having different, path-dependent types does not happen. So we know the exact method definitions that will be called when methods are called on the passed object, because we don't need to check which override is actually being used. This means we can eliminate the overhead of doing virtual method lookup, so the JIT compiler can emit optimized machine code that is often faster than an equivalent C++ call (because in the C++ case, the virtual lookup cannot easily be eliminated).

Java HotSpot VM uses many other techniques to optimize the code that JIT compilation produces. *Loop optimization*, *type sharpening*, *dead-code elimination*, and *intrinsics* are just some of the other ways that Java HotSpot VM tries to optimize code as much as it can. Techniques are frequently layered one on top of another, so that once one optimiza-

tion has been applied, the compiler might be able to see more optimizations that can be performed.

Compilation Modes

Inside Java HotSpot VM, there are actually two separate JIT compiler modes, which are known as C1 and C2. C1 is used for applications where quick startup and rock-solid optimization are required; GUI applications are often good candidates for this compiler. C2, on the other hand, was originally intended for long-running, predominantly server-side applications. Prior to some of the later Java SE 7 releases, these two modes were available using the `-client` and `-server` switches, respectively.

The two compiler modes use different techniques for JIT compilation, and they can output very different machine code for the same Java method. Modern Java applications, however, can usually make use of both compilation modes. To take advantage of this fact, starting with some of the later Java SE 7 releases,

a new feature called *tiered compilation* became available. This feature uses the C1 compiler mode at the start to provide better startup performance. Once the application is properly warmed up, the C2 compiler mode takes over to provide more-aggressive optimizations and,

usually, better performance. With the arrival of Java SE 8, tiered compilation is now the default behavior.

Java HotSpot VM has the ability to produce a more detailed log of compilation events. Let's move on to see how to enable the production of such a log.

Full Logging of JIT Compilation

The switch for enabling full logging is `-XX:+LogCompilation`, and it must

be preceded by the option
`-XX:+UnlockDiagnosticVMOptions`.
 Using the `-XX:+LogCompilation`
 switch produces a separate log
 file, `hotspot_pid<PID>.log`, in the
 startup directory. To change
 the location of the file, use
`-XX:LogFile=<path to file>`.

CHOOSE YOUR MODE

The two Java HotSpot VM compiler modes use different techniques for JIT compilation, and they can output very different machine code for the same Java method. **Modern Java applications, however, can usually make use of both compilation modes.**

LISTING 3

```
<nmethod compile_id='2' compiler='C1' level='3'  
entry='0x000000001023fe240' size='1224'  
address='0x000000001023fe0d0' relocation_offset='288'  
insts_offset='368' stub_offset='880' scopes_data_offset='1032'  
scopes_pcs_offset='1104' dependencies_offset='1200'  
nul_chk_table_offset='1208'  
method='java/lang/String hashCode ()I' bytes='55' count='512'  
backedge_count='8218' iicount='512' stamp='0.350'/>
```



[Download all listings in this issue as text](#)

The output log is a large XML file (often comprising dozens or hundreds of megabytes), containing a high level of detail about the decisions the Java HotSpot VM compilers made. This log contains a lot more information than the simple format we discussed above.

Listing 3 is a sample entry from the detailed compilation log, and it contains a lot of detail about the compilation decisions that Java HotSpot VM made when compiling the method—in this case, the method `String::hashCode()`. However, the format is complex and difficult to work with. This presents a barrier for many developers, which means that they can't use the detailed logs to understand their applications. Fortunately, help is at hand.

In Part 2 of this series, we will introduce JITWatch, a new open source tool that can consume the detailed compilation logs and pro-

vide simple, graphical visualizations of many aspects of JIT compilation. You can download JITWatch from [GitHub](#), which is where continued development of the tool takes place.

Conclusion

In this article, we have introduced some of the basic concepts of JIT compilation as deployed in Java HotSpot VM. We have illustrated the flags needed to produce compilation log output—both the compact format and the more extensive XML output. In doing so, we have paved the way to discuss a new visualization tool in Part 2 of this series. [</article>](#)

MORE ON TOPIC:



LEARN MORE

- GitHub repository for JITWatch

Low-overhead profiling and diagnostics for Java applications running on the JVM

BIO



Java
Virtual
Machine

mainly contains the equivalents of two of the JRockit Mission Control tools: the Java Management Extensions (JMX) Console and the Java

Flight Recorder. There is no online heap analyzer yet. There is, however, a set of quite useful (experimental) plugins that extend Oracle Java Mission Control to do heap dump analysis, do targeted analysis for various Oracle products, or simply extend existing Java Mission Control functionality in more-useful ways.

Getting Started with Oracle Java Mission Control

Starting Oracle Java Mission Control is quite easy. Download and install a recent-enough Java SE JDK (7u40 or later), and then simply run `%JDK_HOME%/bin/jmc`. The alien thing that starts is not, as I am sometimes asked, a native application. It's Java, but it's built upon Eclipse RCP technology. If you would rather run Oracle Java Mission Control inside the Eclipse IDE, you can install it into your Eclipse from the [Oracle Java Mission Control](#) site.

The JMX Console. The console in Oracle Java Mission Control can be thought of as a [JConsole](#) on steroids. As shown in **Figure 1**, it allows you to monitor JMX data in various ways, to take action when attributes attain certain values, and to persist the data and later look at what was recorded. There are various experimental plugins for the console, such as an Oracle

Coherence plugin, a plugin for running JConsole plugins, and a plugin for tweeting messages when an action is triggered.

To connect the console to a JVM, simply choose the JVM process you want to connect to in the JVM browser tree and select **Start JMX Console**. JVM processes will appear automatically in the JVM browser tree if the JVM is started locally or

with the [Java Discovery Protocol \(JDP\)](#). If you have a remote JVM without JDP running, just enable the built-in [jmxrmi](#) agent as [you normally would](#) to be able to connect with JMX clients such as JConsole.

The console is typically used to monitor a small set of critical attributes, such as the CPU load and Java heap usage, that are sampled

at a relatively low frequency. The console can be configured to take action when attributes reach an undesirable value, and one of those actions can be to dump Java Flight Recorder data. The console also contains special tabs for looking at thread information, such as dead-locked threads, per-thread allocation information, and per-thread profiling information. That said, the

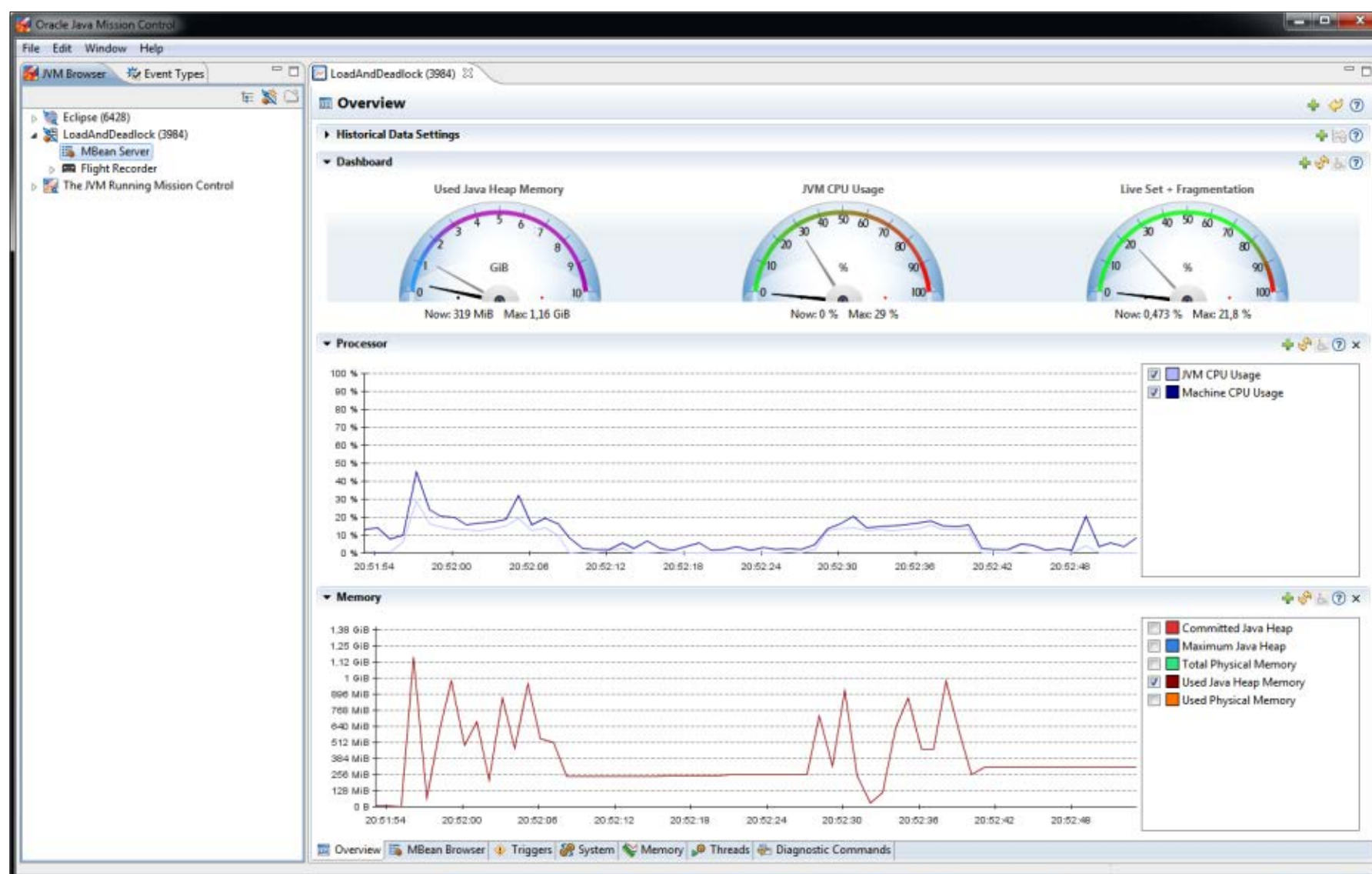


Figure 1

events for various thread-stalling activities that can occur, such as blocking on entering a monitor, parking, waiting, and so on. Latency profiling is usually the first place to look if you do not have a CPU-bound problem, but you still have performance issues.

- OS information. There is a lot of

operating system information, for example, information about CPU load, JVM CPU load, environment variables, and running processes. If you still can't find what you're looking for, Oracle Java Mission Control has a [DTrace plugin](#) for retrieving everything you ever wanted to know but

were too afraid to ask. Note that the overhead for using DTrace, even with very few probes, is usually more than an order of magnitude higher than the overhead for using the Java Flight Recorder, so use with caution.

Much more information is available from the event providers

built into the JVM, such as class loading and compiler events. One way to learn more about what is available is to take a closer look at the [metadata](#) from a recording.

Conclusion

As of JDK 7u40, a new tool suite is bundled with the JDK: Oracle Java Mission Control. The main focus of the suite is on production-time profiling and diagnostics. This focus means that the gathered data is quite true to the dynamics of the application being profiled, because the observer effect is kept quite low. In other words, instead of profiling the profiler itself, most of the time is actually spent profiling the application and the runtime.

While the main focus of Oracle Java Mission Control is production systems, it can be quite useful during development, too. It is also free for use during development, per the standard Oracle Binary Code License (BCL). [</article>](#)

MORE ON TOPIC:



LEARN MORE

- [Oracle Java Mission Control home page](#)

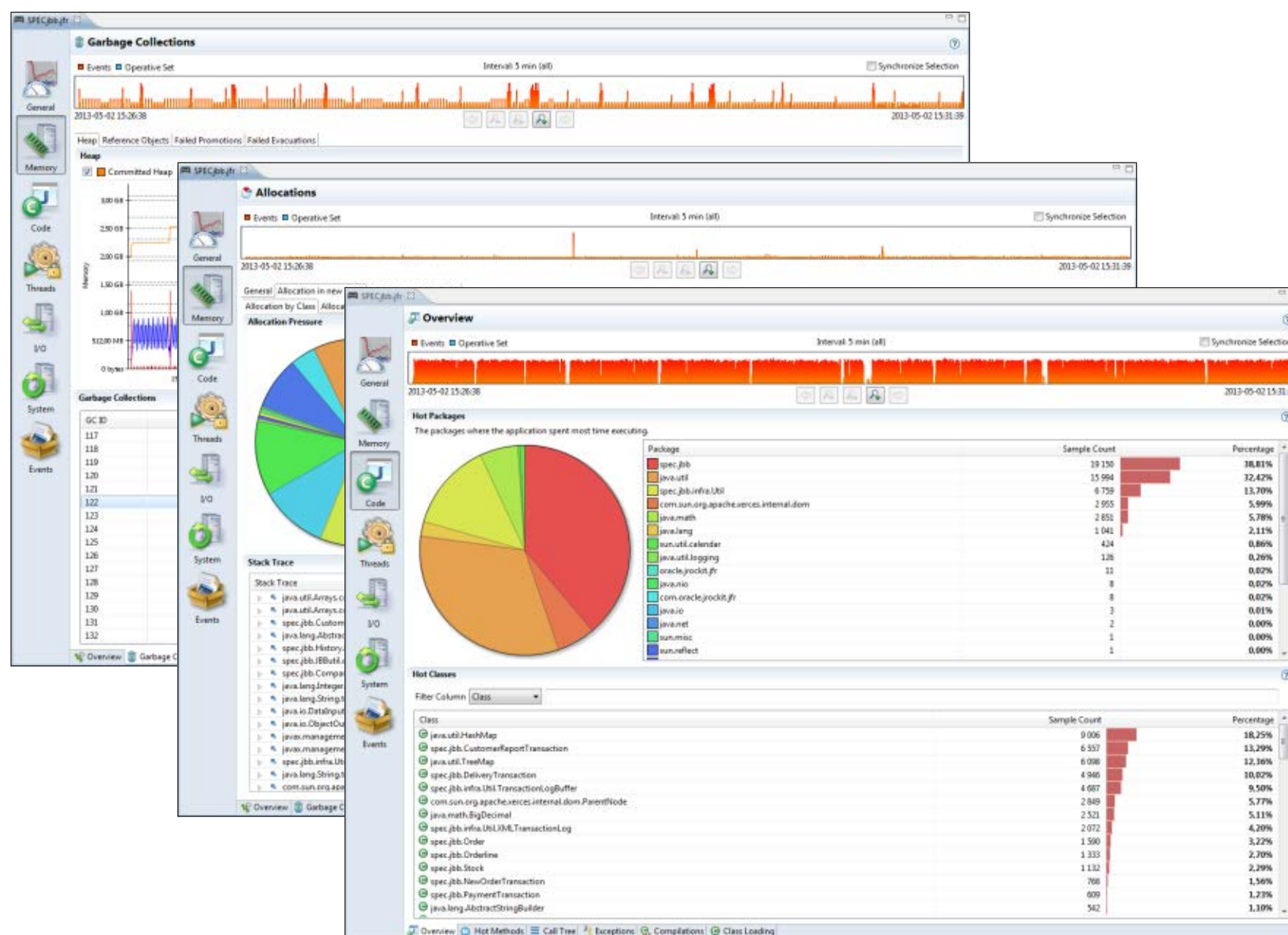


Figure 2

**CASIMIR** SATERNOS

Server-Side Deployment on the JVM

JVM deployments with the server outside, inside, and alongside a web app

Over time, the success of a web application can influence technology choices for ongoing support and customization. For example, Twitter was initially a Ruby on Rails application but the back-end Ruby services were replaced with applications running on the Java Virtual Machine (JVM) and written in Scala. With this in mind, it is important to recognize that Java development practices and the use of the JVM have expanded greatly in recent years. These changes can ease the transition from starter projects to large-scale, mature applications.

This article will demonstrate how a project can initially be developed very quickly with a view toward a variety of deployment options. More specifically, a small web application will be written in Ruby but packaged in a web application archive (WAR) file. Options for

deploying the WAR file with the server outside, inside, or alongside allow such an application the type of scalability and availability demanded in modern high-traffic, cloud-based deployments. By using a standard WAR file for deployment, the application is packaged in a form that can readily be replaced with a similar package written in Java or other JVM language.

Note: For the example described in this article, you can use any Ruby implementation of your choice (either JRuby or a C-based Ruby).

Java and Alternative JVM Languages

The JVM has emerged as the preeminent language-independent virtual machine target and is optimized to run on a wide range of platforms. It has matured over the years and now supports a growing number of alternative languages that compile to JVM

bytecode. Each release of the JVM includes new features that enhance the JVM's ability to support and optimize the performance of additional languages. These qualities have resulted in the popularity of the JVM and make it a serious consideration for many types of software projects.

Java is a fine general-purpose software development language, but it is not necessarily the best option for every project. JVM languages include JRuby, Jython, Groovy, JavaScript, Scala, and Clojure.

These languages are impressive implementations in their own right, and all share the desirable quality of running on the JVM. During the initial phases of a project, it might be preferable to use scripting

languages specifically geared for programmer productivity. By using JVM implementations, a project written in one language can be ported more easily to another, if needed at a later point.

JRuby is one of a number of popular JVM languages in use. It is less “strict” than Java in several ways, which promotes quick initial development. But some of the features that make Ruby a popular choice for programmer productivity can be the source of problems as a project grows. Java

provides type safety and the high performance required by more-mature projects. One way the process of transitioning a project from Ruby to Java can be simplified is by packaging Ruby code in a Java

CREATIVE TYPES

Programmers are creative people and typically **delight in contriving clever ways to solve problems.**



ORACLE.COM/JAVAMAGAZINE //////////////////////////////////// JULY/AUGUST 2014



Introduction to the Java Temporary Caching API

JAVA IN ACTION

implementation details. This article provides an introduction on how to leverage the Java Temporary Caching API in Java applications.

an important area for standardization in 2001. However, it wasn't until March 18, 2014, that the specification was finally released.

applications. Indeed, a cache can also be used to store images, responses from web services, and so on. The most generic representation of an entry that can be cached, as defined by JSR 107, is a Java object.

Sometimes, the transition between the formation of the Expert Group and the final approval of the specification happens very quickly. In other cases, it takes a bit longer. The Java Temporary Caching API, JSR 107, was considered

While caching is often linked to database calls—that is, the result of a database call is often stored in a cache—it should be stressed that caching concepts are much broader than database

Note: The source code for the sample application developed in this article can be downloaded [here](#).

Amazon DynamoDB

Amazon DynamoDB is a cloud-based NoSQL data store provided by Amazon.

JAVA TECH

ABOUT US



ava
net

log



municate with DynamoDB. In our examples, we will use the high-level API, which provides a direct mapping between Java objects and entries in DynamoDB. In order to do so, we will use a few annotations on the Java objects we want to store.

The Sample Application

While most of the applications that will leverage the Java Temporary Caching API are probably enterprise applications, the API itself is designed to work in a Java SE environment as well. For simplicity, we will write our examples on top of the Java SE 8 platform.

Our sample application will create a number of **Person** instances. The code for the **Person** class is shown in **Listings 2a** and **2b**. The **Person** class is a typical Java class containing fields for the first name, the last name, and the age of a person. Additionally, we have a **myKey** field that will be used for storing a primary key.

This command will start the local version of DynamoDB and cause it to run in memory instead of using a database file. As a consequence, all data supplied to DynamoDB Local will be kept in memory and will be lost when the server is stopped. This is by no means how the cloud version of DynamoDB works. However, the APIs required for accessing the local version and the cloud version are the same. The cloud offering requires more configuration and, unlike the local version, is not a free service. Therefore, DynamoDB Local is very popular with software engineers during the code development process.

The Amazon Web Services SDK (AWS SDK) contains a number of APIs that allow developers to com-

The annotation `@DynamoDBTable(tableName = "Person")` is used to associate the `Person` class with the "Person" table that we will create in DynamoDB. Further, the `@DynamoDBHashKey` annotation indicates that the annotated method (`getMyKey`) returns the hash key for the object in DynamoDB.

LISTING 2b

```
java -Djava.library.path=./DynamoDBLocal_lib
-jar DynamoDBLocal.jar -inMemory
```

 [Download all listings in this issue as text](#)

The examples we will create will perform three steps:

1. Create a datastore.
2. Populate the datastore.
3. Query the datastore.

The `main` method for our sample application is very simple (see **Listing 3**).

In our first example, we will not use any caching. We will write all data directly to DynamoDB, and when querying data, we will directly query DynamoDB as well.

The `createDatabase` function will create the DynamoDB datastore. The code in **Listing 4** demonstrates how a DynamoDB datastore is created. We won't go into DynamoDB-specific details, but on a high level, the `createDatabase` call does the following:

- Creates credentials (key and secret) for communicating with Amazon DynamoDB. For the DynamoDB Local version, these credentials don't matter, although they have to be supplied.
- Creates an `AmazonDynamoDBClient` and a `DynamoDBMapper` instance. We will use the `DynamoDBMapper` instance later.
- Defines the key that will be used as the primary key for indexing the data.
- Specifies the requested throughput (for read and write operations). Again, these values are

irrelevant for the DynamoDB Local version, but they have to be provided.

- Deletes the table "Person," if it exists.
- Creates the table "Person."


Now that a table has been created, and a `DynamoDBMapper` object has been constructed, we can populate the table.

We will create 1000 instances of `Person` with random content. An index ranging from 0 to 999 will be used to set the `myKey` field for these instances. The code in **Listing 5** populates the table with these instances. Using the DynamoDB high-level API, storing an object is as simple as calling the `save` method on a `DynamoDBMapper` instance and providing the object that we want to store.

We will now query DynamoDB to ask for **Person** instances with a specific key, as shown in **Listing 6**. We will make 1000 requests to the **DynamoDBMapper**. Each request asks the **DynamoDBMapper** to find the **Person** instance corresponding to a specific key. The key is a random value between 0 and 1099, whereas the keys of the stored **Person** instances range between 0 and 999. As a consequence, we expect about 10 percent of the requests to result in an empty answer.

LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6

```
public static void main(String[] args) {
    createDatabase();
    populateDatabase();
    queryDatabase();
}
```

 [Download all listings in this issue as text](#)

Introducing the Java Temporary Caching API

Until now, all requests for storing and retrieving data have gone directly to the DynamoDB system. At this point, we will introduce the Java Temporary Caching API. Before we look at the code, here are the main classes defined by JSR 107:

- **CacheProvider**, which contains an implementation of the Java Temporary Caching API
 - **CacheManager**
 - **Cache**
 - **Entry**
 - **Expiry**
- We will now modify the code used to populate the datastore

and introduce the caching features. The modified code is shown in **Listing 7**. Before we populate the datastore, we create a **Cache** instance. First, we have to obtain a reference to the **CachingProvider**. This is done by the code shown in **Listing 8**.

The Java Temporary Caching API is defined by JSR 107, but the API itself does not contain a concrete implementation. At runtime, the API will use the Java [ServiceLoader](#) to check whether a [CachingProvider](#) implementation is available on the classpath.

In our example, we use the Reference Implementation of JSR 107, which is available in Maven Central. The JAR file containing the Reference Implementation contains a **META-INF/services** directory containing a reference to the **CachingProvider** implementation.

If another JSR 107-compliant implementation is available on the classpath, that implementation will be used. Our code doesn't have to change, though, because we are using only the APIs provided by the JSR 107 specification. We are not

COMMON APPROACH

The Java Temporary Caching API allows Java developers to use a common approach for working with caches, **without having to worry about implementation details.**

using implementation-specific APIs or features.

Once we have a `CachingProvider`, we need to get a `CacheManager`. This is done using the code shown in **Listing 9**.

Now that we have a `CacheManager` instance, we can create a `Cache` instance. Our cache needs to hold instances of `Person`. A `Cache` contains entries that have a key and a value. In our case,

it makes sense to use the `myKey` field of the `Person` as the key and to use the `Person` itself as the value. The code in **Listing 10** creates our cache and provides the required configuration.

The configuration of the cache is defined in a `MutableConfiguration` instance. The `MutableConfiguration` class allows you to set a number of configuration policies by using the fluent API . The fluent API implies that the result of a `set` method is the instance itself.

In our example, we set the following:

- The `setTypes` statement in **Listing 10** specifies that the key for our cache is of type `Long`, and the value is of type `Person`.

LISTING 7

LISTING 8

LISTING 9

LISTING 10

```
private static void populateDatabaseCache() {

    CachingProvider cachingProvider =
        Caching.getCachingProvider();
    CacheManager cacheManager =
        cachingProvider.getCacheManager();

    MutableConfiguration<Long, Person> config
        = new MutableConfiguration<Long, Person>()
            .setTypes(Long.class, Person.class)
            .setExpiryPolicyFactory(
                AccessedExpiryPolicy.factoryOf(ONE_HOUR))
            .setStatisticsEnabled(true);

    cache = cacheManager.createCache("personCache", config);

    Random age = new Random();
    DynamoDBMapper mapper = new DynamoDBMapper(client);
    for (long i = 0; i < 1000; i++) {
        String f1 = UUID.randomUUID().toString();
        String f2 = UUID.randomUUID().toString();
        Person person = new Person(
            i, f1, f2, age.nextInt(100));
        mapper.save(person);
        cache.put(person.getMyKey(), person);
    }
}
```



[Download all listings in this issue as text](#)

- The `setExpiryPolicyFactory` statement defines how long cache entries are valid.

In order to create a `Cache` instance, we call the `createCache` method on the `CacheManager` instance and supply the name of the cache (`personCache`) and the

configuration object.

Apart from creating the cache, we add only one line of code. In the `for` loop where the `Person` instances are created, we add the following line:

```
cache.put(person.getMyKey(),
person);
```


and we map the methods in this **CacheWriter** to API calls to DynamoDB. Each time an entry is added to, updated in, or deleted from the cache, the **CacheWriter** methods are called. As a consequence, we can execute all storage tasks on the cache only. Thanks to the **CacheWriterFactory**, changes in the cache will be written to DynamoDB as well.

Similar to a `CacheWriterFactory`, we can also specify a `CacheLoaderFactory`. The `CacheLoader` that is returned by the `CacheLoaderFactory` is responsible for querying DynamoDB if an entry is requested from the cache but is not found there. In this case, the overridden implementations from the `CacheLoader` will send queries to DynamoDB.

The code in **Listing 12** adds the read-through behavior on the `MutableConfiguration` instance we used before. As you can see, the `load` method in the `CacheLoader` does exactly what we did in **Listing 6**. It creates a query for DynamoDB and tries to retrieve the `Person` using the provided key.

Using the `CacheLoader`—and, hence, leveraging the read-through behavior—often makes it easier for developers to focus on the application logic rather than on the cache logic. Without read-through, a

developer has to query the persistent storage every time a query on a cache returns no result. By using a **CacheLoader**, that behavior is delegated to the **CacheLoader**, and it needs to be coded only once.

Conclusion

In this article, we only scratched the surface of JSR 107. The specification provides a common approach to functionality offered by most caching providers. We covered only simple get and store operations, and we briefly

touched on the concepts of write-through and read-through. There is much more to discover, and the interested reader is referred to the Javadoc for more information.


The most important achievement of JSR 107 is that application developers can now use a caching strategy in their applications, independent of a specific implemen-

NOT JUST JAVA EE

While most of the applications that will leverage the Java Temporary Caching API are probably enterprise applications, **the API itself is designed to work in a Java SE environment as well.**

LISTING 12

```
config.setReadThrough(true)
config.setCacheLoaderFactory(new Factory<CacheLoader<Long, Person>>() {
    public CacheLoader<Long, Person> create() {
        return new CacheLoader<Long, Person>() {
            public Person load(Long k) throws CacheLoaderException {
                Person hashKeyValues = new Person();
                hashKeyValues.setMyKey(k);
                DynamoDBQueryExpression<Person> queryExpression = new Dynam
oDBQueryExpression<Person>().withHashKeyValues(hashKeyValues);
                List<Person> itemList = mapper.query(Person.class, queryExpression);
                if (itemList.size() == 0) {
                    return null;
                } else {
                    return itemList.get(0);
                }
            }
        }
    }
    public Map<Long, Person> loadAll(Iterable<? extends Long> itrbl)
throws CacheLoaderException {
        Map<Long, Person> answer = new HashMap<>();
        itrbl.forEach((Long k) -> answer.put(k, load(k)));
        return answer;
    }
});
}
```

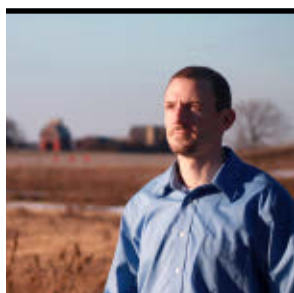
 [Download all listings in this issue as text](#)

tation. Some providers of cache software already have a JSR 107–compliant implementation, and it is expected that more will follow. During development, developers are encouraged to use only the functionality provided by JSR 107. At runtime, a concrete implementation (free or commercial) is added

to the classpath, and it will automatically be used. **</article>**

LEARN MORE

- [Amazon DynamoDB](#)
- [JSR 107](#)
- [Specification and Reference Implementation](#)



JOSH JUNEAU



JavaFX with Alternative Languages

Alternative languages and custom APIs enable rapid JavaFX application development.

In this day and age, the more tools you have in your toolbox, the better. That is, it makes sense to know more than one way to implement a solution, because in some cases one implementation might have advantages over others. Although the JavaFX 8 API is very nice, there are

advantages to knowing more than one way to develop a JavaFX application.

This article demonstrates how to develop JavaFX applications using alternative languages, so that you can add some additional tools to your toolbox. In the first section of this article, we'll

take a look at a JavaFX drawing application, and we will compare its implementation across a couple of different languages. In the second section, we will take a look at GroovyFX, a JavaFX API for the Groovy language, which is focused on rapid JavaFX application development.

Note: The source code for this article can be found on [GitHub](#).

and reset the canvas to do it all over again. The application contains two **ChoiceBox** nodes for choosing the color and size of the pen, a **Canvas** to draw upon, and a **Button** to reset the canvas. **Figure 1** shows what the application looks like when it is executed (and a great artist has spent some time sketching).

Take a look at **Listing 1**, which contains the complete code for the application, written using the JavaFX 8 API. As with all JavaFX applications, it is launched via the execution of a `main()` method, and the `start()` method contains the primary stage construction and much of the application implementation.

To note a few pieces of code in particular, let's first take a look at the **Reset** button event handler, because it is implemented using a lambda expression. When the user clicks the **Reset** button, the canvas is cleared.

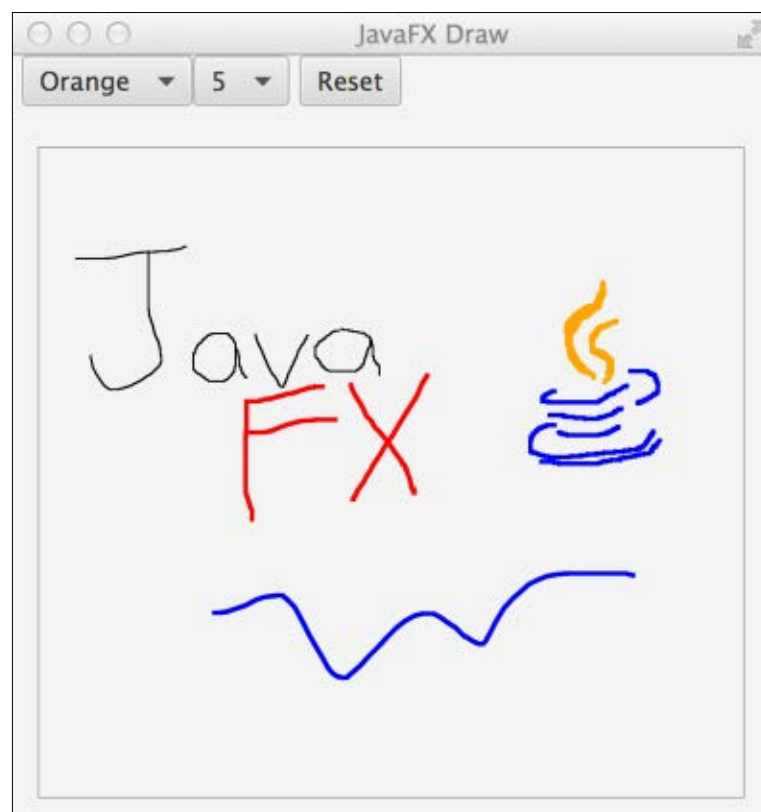


Figure 1



for the **ChoiceBox** selection uses an **if** statement instead:

```
if idx == 0:
    new_color = Color.BLACK
elif idx == 1:
    new_color = Color.BLUE
elif idx == 2:
    new_color = Color.RED
```

If you are new to Python, you'll see that there is a standard indentation structure to which you must adhere. That said, the code is concise and easy to follow. Another note about Jython is that, just as in Groovy, there is no need to declare types, so function parameters and variables are typeless.

Using Custom APIs to Develop JavaFX Applications

We've already seen that implementing JavaFX applications with alternative languages can have advantages. While there are some advantages to coding with alternative languages, there are even more advantages to using an API that is tailored specifically for use with JavaFX. In this section, we will take a look at the GroovyFX API, which

completely changes the way in which a developer can construct a JavaFX application. The developers of GroovyFX have made it easy to visually see the scene graph as it is being built, making code easier to read, understand, and visualize.

GroovyFX at a glance. GroovyFX is an API that allows you to construct a JavaFX application using the Groovy builder syntax. The resulting code is very clean and easy to read, making the construction of JavaFX applications very easy. GroovyFX also eliminates lots of boilerplate code, allowing you to concentrate on what you'd like to get done, rather than on worrying about specific API calls. This is possible because GroovyFX takes advantage of Groovy's powerful domain-specific language (DSL) features and abstract syntax tree (AST) transformation.

Let's begin by transforming the application that was written in the first section of this article using Groovy and Jython so it adheres to the GroovyFX API. **Listing 4** contains the code for the `GroovyFXDraw` application. The code is significantly shorter than any of the other code we've looked at thus far. It is

LISTING 4

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
import static groovyx.javaafx.GroovyFX.start
import groovyx.javaafx.beans.FXBindable
import javafx.stage.Screen
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color
import javafx.collections.FXCollections
import javafx.scene.canvas.Canvas
import javafx.scene.canvas.GraphicsContext
import javafx.stage.Screen

start {
    stage(title: 'GroovyFX Draw', visible: true) {
        scene = scene(id: "sc", fill: WHITE,
            width: Screen.getPrimary()
                .getVisualBounds().getWidth(),
            height: Screen.getPrimary()
                .getVisualBounds().getHeight()) {

            canvas = canvas(id: "drawcanvas",
                width: bind(sc.width()),
                height: bind(sc.height()))
            GraphicsContext graphicsContext =
                canvas.graphicsContext2D
            canvas.onMousePressed {
                MouseEvent event->
                graphicsContext.beginPath()
                graphicsContext.moveTo(event.getX(),
                    event.getY())
                graphicsContext.stroke()
            }
        }
    }
}
```

 [Download all listings in this issue as text](#)

Another significant difference is the way in which the application is launched, because no `main` method is defined. Instead, a Groovy closure that contains the application constructs is simply passed to the `static groovyx.javafx.GroovyFX.start` method to initiate the application:

```
start {  
    // Declare JavaFX  
    // scene graph nodes  
}
```

An additional important difference is the way in which the code is constructed. Each node embedded into the scene graph is constructed as a Groovy closure, and in some cases nodes are embedded within each other. This visual syntax provides an easy way to see how the user interface will be laid out. The properties of each node are also specified differently with GroovyFX, because they are treated as key/value pairs. Each node can have its own set of properties specified by placing them within a comma-separated key/value format, as seen here:

```
hbox(spacing:10, padding:10) {  
  ...  
}
```

There are a couple of easy steps you need to take before you can use GroovyFX. To use GroovyFX in any environment (without the use of an IDE), first, download GroovyFX using the latest distribution from the GroovyFX site. Alternatively, you can build the GroovyFX Project from the source code by cloning the project from GitHub.

To clone the project, ensure that you have [Git](#) installed on your system, and then clone the project by executing the following command:

```
git clone git://github.com/
groovyfx-project/groovyfx.git.
```

Note: Please ensure that you are using GroovyFX 0.4.0 for Java 8 and JavaFX 8 compatibility. The examples for this article were written with GroovyFX 0.3.1 and Java SE 7, as GroovyFX 0.4.0 was not available at the time.

Once you've cloned the project, you can build it from source code using [Gradle](#). The current release, 0.4.0, includes support for JavaFX 2.2 on Java 7 or JavaFX 8 on Java 8.

```
git clone -b O_4_SNAPSHOT git://  
github.com/groovyfx-project/  
groovyfx.git
```

```
cd <<groovyfx_location>>
gradlew build
```

You can then run any of the packaged examples using the **gradlew** command, followed by the name of the example. For example, to run **HelloWorldDemo**, use the following command:

If you prefer to use an IDE such as NetBeans, simply clone the GroovyFX project and open it within NetBeans. By default, the GroovyFX project distribution is a NetBeans IDE project. You can run any of the demos within the NetBeans IDE by expanding the **Demo Programs** folder, right-clicking the demo, and choosing **Run**. There are some excellent GroovyFX examples contained in the distribution. If you wish to use an IDE other than NetBeans, you can refer to the [GroovyFX Guide](#) for details.

Building a basic GroovyFX application. As mentioned previously, all GroovyFX applications are launched using the static `groovyx.javafx.GroovyFX.start` method. The scene graph is passed to the start method by enclosing it within brackets. We're able to use the Groovy builder syntax, but behind the scenes, GroovyFX `SceneGraphBuilder` is used to construct the scene graph.

Each of the JavaFX controls is

available to GroovyFX, but the syntax is a bit different than usual. For instance, each of the controls when used in Java code begins with a capital letter. However, its GroovyFX counterpart must begin with a lowercase letter.

Another difference is that instead of using Java “setters” to configure different control properties, GroovyFX allows you to pass a map (key/value pairs) of different properties to be set. Take, for instance, the JavaFX **Button** control. In JavaFX, the control must be instantiated and then set to different properties, and setter methods must be called upon the **Button** instance. In GroovyFX, a **Button** control is instantiated by placing it within the **Scene** closure, and different properties are set by enclosing them as key/value pairs within parentheses, as follows:

```
button("Save",
    style:"-fx-font: 14 arial",
    onAction: {
        textEntry =
            text.getText()
            println textEntry
    })
```

Note that the event handler is written inline as a Groovy closure, and it is assigned to the `onAction` property. Styles are set inline as

well using the `style` property. You can also access the `javafx.scene.text.Font` directly as a node property. For example, the `style` property from the previous example could be changed to `font: '14pt arial'`, as follows, instead of listing out the full style string:

```
button("Save",
    font: '14pt arial',
    onAction: {
        textEntry =
            text.getText()
            println textEntry
    })
```

The code in **Listing 5** includes this **Button** control, along with a **TextField** and **Label**. The basic application allows you to enter some text into the **TextField**, click the button, and have the text printed to the server log file. It is a very simple application, but you can see that it demonstrates basic layout with GroovyFX, and it uses an **HBox**. In this example, the **HBox** contains the label and the **TextField**, with a spacing of 5.

```
hbox(spacing: 5){
  ...
}
```

In addition, colors and paints can

be set using either the JavaFX [Paint](#) or [Color](#) objects, or via pseudo color variables ([RED](#), [ORANGE](#), [GREEN](#), and so on). Colors can even be defined as a hex string or a JavaFX cascading style sheet (CSS) style, making GroovyFX very flexible. GroovyFX includes a multitude of handy shortcuts; to learn about them, I recommend taking a look at the GroovyFX documentation.

Abstracting inline code.

Additional code can be placed outside the scene graph, and it will still reside within scope for use via scene graph nodes. **Listing 6** demonstrates an example of placing code outside the scene graph in an effort to organize. The code in **Listing 6** is for

the **GroovyFXDraw** application, but the **Button** control for resetting the **Canvas** has had its event handler code abstracted and placed within a closure that has been defined outside the scene graph.

This style of coding for GroovyFX applications can come in very handy when working with large scene graphs that contain many events. It also can be handy if you want to make a class accessible within your scene graph, but still place it within the same Groovy file.

LISTING 5

LISTING 6

```
import static groovyx.javafx.GroovyFX.start

start{
    def textEntry;
    stage(title: "GroovyFX Introduction", width: 600,
        height: 300, visible: true){
        scene(fill: groovyblue){
            hbox(spacing: 5){
                label("Place Text Here:",
                    textFill: white,
                    halignment: center,
                    valignment: center)
                text = textField(promptText: "Type here",
                    prefColumnCount: 25)
            }
            button("Save", style:"-fx-font: 14 arial",
                onAction: {
                    textEntry = text.getText()
                    println textEntry
                })
        }
    }
}
```



Binding properties with GroovyFX.

One of the most powerful features of JavaFX is its binding ability. Binding allows nodes within a scene graph to exchange data and react together in real time. For instance, if the `text` property of a `TextField` control is bound to a

`Label`, then the `Label` text will reflect any changes that are made to the `TextField` contents as a user is typing. JavaFX binding can contain lots of boilerplate code, and GroovyFX reduces the boilerplate by using its `bind` utility. The following `Label` written in GroovyFX demonstrates



Figure 2

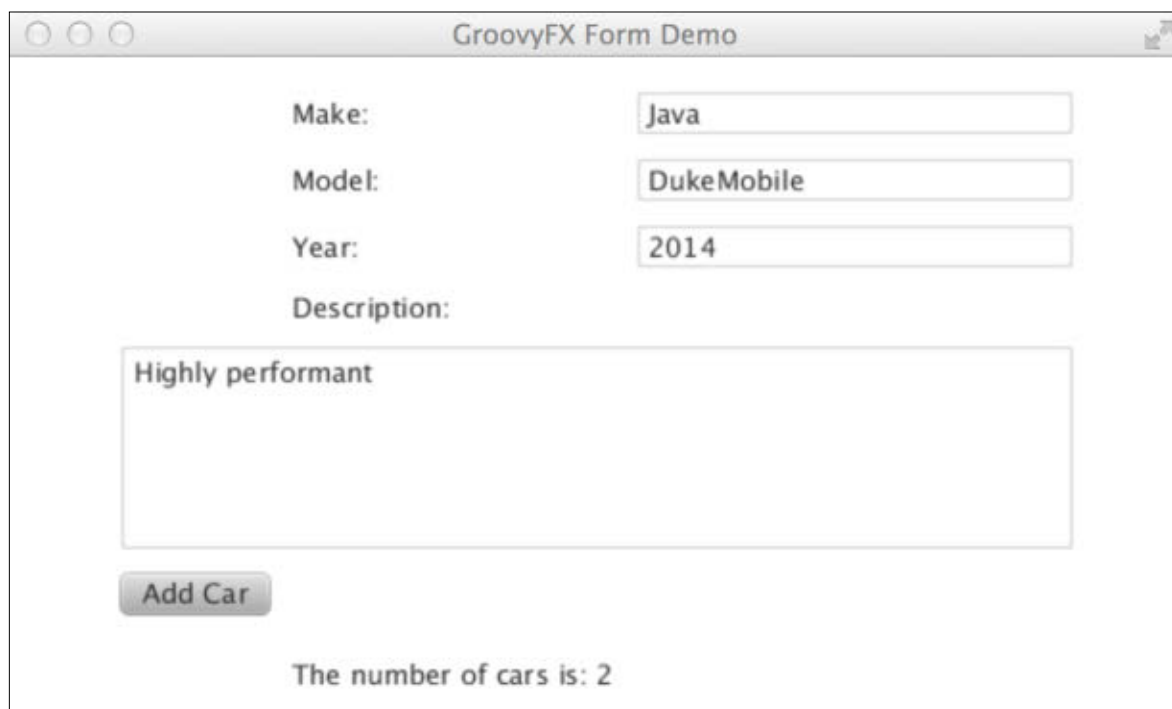


Figure 3

binding to a `TextField` named `text`.

```
Label(text:bind(text, 'text'))
```

The code in **Listing 7** demonstrates simple binding using this technique. When the code is executed, the **Label** will mirror any text typed into the **TextField** (see **Figure 2**).

GroovyFX also provides a convenience annotation, `@FXBindable`,

to automatically mark a property or entire class as bindable. By applying the `@FXBindable` annotation to a property or class, the binding logic will be automatically added to the code at runtime, alleviating the need to write redundant boilerplate code.

In the next example, a simple form has been generated for entering car information (see **Figure 3**). The form fields are each bound to

LISTING 7

```
String textEntry
start{

stage(title: "GroovyFX Introduction", width: 600,
height: 300, visible: true){
scene(fill: groovyblue){
gridPane(hgap: 5, vgap: 10,
padding: 20){
hbox(spacing: 5, row:1,
columnSpan:2 ){
label("Place Text Here:",
textFill: white,
halignment: center,
valignment: center)
text = textField(
promptText: "Type here",
prefColumnCount: 25)
}

button("Save", font: "14pt arial",
row: 2, column: 1,
onAction: {
textEntry = text.getText()
println textEntry
})
label(text: bind(text, 'text'),
row: 3, column: 1)
}
}
}
}
```

 [Download all listings in this issue as text](#)

fades from 0 opacity to 1 over a specified time span.

To see the code for the animation example, see **Listing 9**.

FXML support. Placing view code into FXML files is a recommended approach for separating view code from business logic for JavaFX applications. GroovyFX supports the use of FXML by referencing the files via an `fxml` node within the scene graph:

```
scene {
  fxml(new File('./myfxml.fxml')
    .text)
}
```

Other Alternative APIs

Other alternative languages, such as JRuby and Scala, can be used to develop JavaFX applications as well. In fact, these two languages, in particular, have their own specialized APIs for JavaFX development: JRubyFX and ScalaFX, respectively. These APIs are similar to GroovyFX in that they provide an easy way to work with the JavaFX API in the alternative language syntax. For more information, check out their project websites.

TAILOR-MADE

Implementing JavaFX applications with alternative languages can have advantages. There are even more advantages to using an API that is **tailored specifically for use with JavaFX**.

Conclusion

JavaFX 8 contains a bevy of new features, and the ability to make use of Java SE 8 constructs, such as lambdas and streams, makes JavaFX development even easier. Alternative languages can provide benefits over—and in some cases work hand in hand with—the JavaFX API. In some cases, alternative languages have their own DSLs for developing JavaFX, which can be even more productive and easy to use.

Add a new tool to your JavaFX toolbox today by learning an alternative approach to developing JavaFX applications. [</article>](#)

MORE ON TOPIC:




LEARN MORE

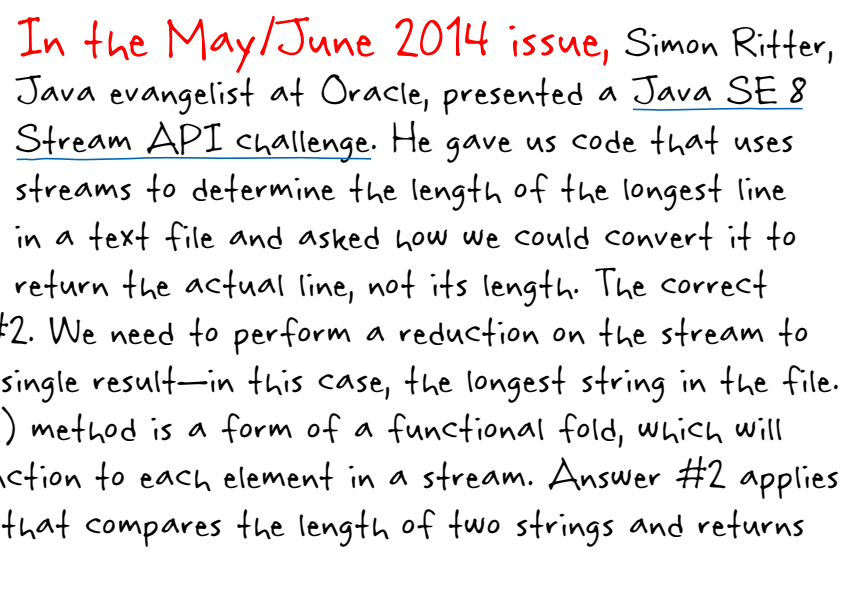
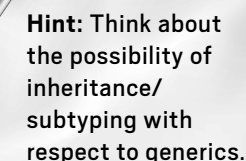
- ["Introducing GroovyFX: It's About Time"](#)
- [Jython website](#)

LISTING 9

Note: The following listing has been excerpted for space, as noted by the ... symbol. The full code listing is available by downloading the code listings for this issue.

```
import static groovyx.javaafx.GroovyFX.start
start {
    stage(title: "GroovyFX YoYo", width: 500,
        height: 400, visible: true) {
        scene(fill: GROOVYBLUE) {
            circle = circle(centerX: 250,
                centerY: 60,
                radius: 60,
                fill: WHITE,
                stroke: BLACK){
                effect: boxBlur(10, 10, 3)
            }
            line = line (startX: 250,
                endX: 250,
                startY: 0,
                endY: 0,
                strokeWidth: 3)
            myTxt = text("Groovy!",
                x: 200, y: 150, fill: ORANGE,
                font: "bold 26pt Arial"){
                fade = fadeTransition(
                    duration: 5000.ms,
                    fromValue: 0.0,
                    toValue: 1.0)
                effect dropShadow(offsetY: 4)
            }
        }
    }
}
```

 [Download all listings in this issue as text](#)



This issue's challenge comes from Abhishek Gupta, a senior identity and access engineer, who gives us a generics problem.

In spite of the introduction of generics in Java SE 5, raw types are supported in order to provide backward compatibility for code written using JDK 1.4 and below. But what happens when we mix generic and raw types?

Assume that this code is compiled and executed with JDK 1.5 or above, which has support for generics.

3 WHAT'S THE FIX?

- 1) The class compilation fails because of an attempt to add a primitive int type to a List.
- 2) The compilation fails and is resolved by changing the signature of the add method to use a generic type (List<Object>) instead of a raw List.
- 3) Compilation is successful but there is a java.lang.ClassCastException at runtime. It is fixed by changing the signature of the add method to use generics and invoking the add method using the appropriate argument type.
- 4) Compilation is successful but there is a java.lang.ClassCastException at runtime. It is fixed by changing the signature of the add method to use a generic type (List<Object>) instead of a raw List.

Look for the answer in the next issue. Or submit your own code challenge!